Automated Theorem Proving for Mathematics: Real Analysis in PVS

A thesis submitted to the
UNIVERSITY OF ST ANDREWS
for the degree of
DOCTOR OF PHILOSOPHY

by

Hanne Gottliebsen

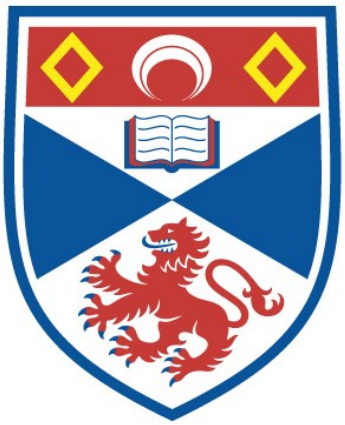
School of Computer Science
University of St Andrews

July 2001

AUTOMATED THEOREM PROVING FOR MATHEMATICS : REAL ANALYSIS IN PVS

Hanne Gottliebsen

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews



2002

"If we knew what it was we were doing, it would not be called research, would it?"

— Albert Einstein

Abstract

Computer Algebra Systems (CASs), such as Maple and Mathematica, are now widely used in both industry and education. In many areas of mathematics they perform well. However, many well-established methods in mathematics, such as definite integration via the fundamental theorem of calculus, rely on analytic side conditions which CASs in general do not support.

This thesis presents our work with automatic, formal mathematics using the theorem prover PVS. Based on an existing real analysis library for PVS, we have implemented transcendental functions such as exp, cos, sin, tan and their inverses, and we have provided strategies to prove that a function is continuous at a given point. In general, this is undecidable, but using certain restrictions we can still provide proofs for a large collection of functions. Similarly, we can prove that a function has a limit at a point. We illustrate how the extended library may be used with Maple to provide correct results where Maple's are incorrect.

We present a case study of definite integration in the CASs axiom, Maple, Mathematica and Matlab. The case study clearly shows that apart from axiom the systems do not fully check the necessary conditions for the definite integral to exist, thus giving results varying from plain incorrect to correct, even if the latter is difficult to detect without manipulating the result.

The extension and correction of the PVS library consists of around 1000 theorems proven by around 18000 PVS proof commands. We also have a test suite of 88 lemmas for the automatic checks for continuity and existence of limits.

Thus we have devised and tested automatic computational logic support for the use of formal mathematics in applications, particularly computer algebra.

I, Hanne Gottliebsen, hereby certify that this thesis, which is approximately 44000 words in length, has been written by me, that it is the record of work carried out by me, and that it has not been submitted in any previous application for a higher degree.

date 20/11/2001 *signature of candidate* _____

I was admitted as a research student in February 1998 and as a candidate for the degree of Doctor of Philosophy in February 1999; the higher study for which this is a record was carried out in the University of St Andrews between 1998 and 2001.

date 20/11/2001 *signature of candidate* _____

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Doctor of Philosophy in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

date 26/11/2001 *signature of supervisor* _____

In submitting this thesis to the University of St. Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and abstract will be published, and that a copy of the work may be made and supplied to any *bona fide* library or research worker.

date 20/11/2001 *signature of candidate* _____

Acknowledgements

I would like to thank my supervisor, Ursula Martin, for her continued help and encouragement throughout this research. I have also gained from discussions and advice from friends and colleagues: Andrew Adams, Martin Dunstan, Roy Dyckhoff, Niels O. Jensen, Tom Kelsey, Steve Linton, Gordon Milligan and Álvaro Rebón.

I am grateful to Bruno Dutertre for allowing me to use, and make changes to, his PVS real analysis library and for the enlightening discussions we have had about our research. I also wish to thank John Harrison for giving me access to his HOL Light implementation of real analysis.

I would also like to give my thanks to the PVS developers at SRI International for hosting my visit there during the spring of 2000 and for their support and many lively discussions and suggestions. In particular thanks goes to John Rushby and Natarajan Shankar for their suggestions regarding automation and Dave Stringer-Calvert and Sam Owre for more technical help. A very special thank you goes to Sam for a seemingly endless stream of patches to give me the facilities I wanted in PVS. Thanks also to the PVS help mailing list, from which I have had many swift and thorough answers and always in a very friendly manner.

Thanks also to some friends who helped make St Andrews home for me: Sandra Brandie and Graham Armstrong from 8^{th} Fife St Andrews Scout Group, and Helen Blair-Rains, Juliet and Alastair Curry, and Mark Clifford from St Andrews Baptist Church.

Finally, I would like to thank my parents and my brother for their continued support throughout my studies. Their encouragement has been invaluable.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis describes our research on automatic computational logic support for formal mathematics, in particular its applications to computer algebra. We give the background and motivation for our research and describe our implementation addressing some of the issues identified. We have used the specification and verification tool PVS [OSRSC99b] for our implementation.

In Section 1.1 we briefly describe our aims and motivation, this is then further elaborated in Chapter 2. Section 1.2 contains a brief description of our achievements, which are explained in detail in chapters 5 and 6. Finally, Section 1.3 contains an overview of this thesis.

Throughout this thesis we attempt to use standard terminology and notation with regard to mathematics, computer algebra and theorem proving.

1.1 Aims and Motivation

The motivation for our research originates from the unreliability of computer algebra systems (CASs) and the idea that computational logic can be used to alleviate some of these problems. CASs work within algebra, using algebraic definitions, theorems and methods. However, many mathematical theorems and methods rely on analytic side conditions, which CASs do not handle correctly. This may lead to ambiguous or even erroneous results.

The aim of our research is to explore and identify areas within mathematics as used by CASs where computational logic can be used to support CASs. Having identified such areas, we aim to implement this support in the theorem prover PVS with a high level of automation in order to shield the CAS user from the technicalities of using a theorem prover.

Bernardin [Ber96] reviews the abilities of different CASs to solve a certain set of equations. The approach is to give each tested system a score depending on the result they give. A higher score is given for returning even partially correct answers: i.e. correct only in special cases. This scoring system favours CASs such as Maple, which often returns answers that are only correct under additional assumptions. Conversely, axiom takes a more careful approach and tries only to give results known to be true, thus axiom gets comparatively low scores in this test.

As a contrast to CASs, which are concerned with calculations – and often with the emphasis on providing *some* answer rather than necessarily a *correct* one – we have formal mathematics. Formal mathematics is about doing mathematics within some computational logic system, whether it is purpose-built to do mathematics, such as Automath [Rez83] or Mizar [Rud92], or more general systems, such as theorem provers like PVS [OSRSC99b] or HOL [GM93]. Thus in formal mathematics, we want to be *certain* that the answers are correct, even if this sometimes means that we can not handle as large a class of problems as CASs.

Since we aim to use formal mathematics to support CASs, we need to ensure that the implementation can be used fairly automatically. In general, theorem provers are quite difficult to use, however, they also support construction of application specific automation. This allows developers to shield end-users from the details of using theorem proving. Thus we can improve the certainty of CAS results without making the CAS any more difficult to use.

1.2 Results and Achievements

We conducted a case study of symbolic definite integration in the CASs axiom, Maple, Mathematica and Matlab. This showed that even for fairly simple definite integrals with

parameters some CASs often returned only partially correct, or even incorrect, results.

We chose to use the theorem prover PVS [OSRSC99b] for our implementation of support for automatic formal mathematics. This choice was based on practical considerations: the real numbers are built into PVS, Dutertre had already implemented some basic real analysis [Dut96] in PVS, and PVS has a high level of automation built-in together with a powerful strategy language for developing further automation. Another possible choice was Harrison's HOL Light, which is a variation of the theorem prover HOL. Harrison had already implemented a real analysis library in HOL Light, however we chose not to use this as HOL Light is not a main stream theorem prover and the continued support for the system is not as certain as that of PVS. We based our implementation on Dutertre's real analysis library and we identified and corrected an error in the definition of limits of functions. This change in the definition reflected on theorems in his development, some of which had to be proved again and some of which were not true with the new definition.

We have implemented transcendental functions, such as exp, cos, sin, tan and their inverses. The functions exp, cos and sin are defined by their power series, and we have proven a large collection of lemmas about properties of the functions, such as various identities involving trigonometric functions. This can then be used in further development and verification within PVS, or indeed by other systems, provided an interface to PVS is available.

We have also developed automation to be used with our library of transcendental functions. The automation is based on type judgements added to the PVS theories and then used for typechecking and matching during proofs. This then allows the built-in strategies, such as `grind`, to search for proofs.

One of the areas of mathematics we identified where CASs could benefit from formal mathematics support is in checking if functions are continuous, and using our automation, we can prove that functions such as

$$f(x) = \frac{5x^2}{\exp(\cos(ax^3 + b))}$$

are continuous with a single proof command `cts`. Likewise, we can prove that similar functions have limits at particular points.

Our library of transcendental functions is used with the Maple-PVS interface developed by our group at University of St Andrews. Work is ongoing developing applications within

Maple using this library.

The library contains about 1000 theorems and lemmas above what Dutertre implemented, and a total of about 18000 PVS proof commands. There are 9 new proof strategies of which 4 were only used in early experiments.

1.3 Thesis Structure

In Chapter 2 we discuss the motivation for our research further, in particular we outline some issues relating to correctness of CASs. We also give a brief introduction to higher order theorem proving and describe some projects on formalising mathematics. Finally, we discuss various approaches to combining computer algebra systems and theorem provers, and describe a prototype interface between the CAS Maple and the theorem prover PVS which uses our library of transcendental functions.

Chapter 3 describes a case study of symbolic definite integration in four different CASs: axiom, Maple, Mathematica and Matlab. The results may be incorrect, correct under certain assumptions or correct. Some of the systems use look-up tables to provide answers in some cases, and common for all the systems is that they make assumptions on the input without informing the user of those assumptions. All the examples in the case study are fairly simple, yet as many of the CAS results are not generally correct, they are obviously difficult enough to demonstrate that symbolic definite integration in CASs is in need of improvements.

In Chapter 4 we describe the theorem prover PVS with respect to the special properties we utilise in our implementation of transcendental functions with automation. We give an overview of the real numbers in PVS and Dutertre's real analysis library. We also explain how PVS's notion of type judgements, which, just like lemmas, are written into theories and then proven, may be used to greatly enhance PVS's built-in proof commands by enabling matching which would otherwise not happen automatically. Finally, we discuss how we can use PVS's strategy language to write new strategies to provide new high level proof commands.

Chapter 5 gives an overview of our implementation of a library of transcendental functions in PVS. This is based on Dutertre's existing real analysis library and Harrison's develop-

ment of transcendental functions in HOL Light, of which we give a brief description. We then explain how we develop transcendental functions from power series. Having defined sine and cosine and proven various well-known facts about them, we then define π as *two times the first positive zero of cosine*, just as Harrison does. This then allows us to prove that $3 < \pi < 3.2$, which so far has been an adequate bound for our needs. As tangent is undefined at all odd multiples of $\frac{\pi}{2}$, and PVS is strongly typed, we then define a new type

```
cos_nz_type: NONEMPTY_TYPE = {x | FORALL k: x /= (2 * k + 1) * pi / 2}
```

which is to be the domain type for tangent. We then define tangent and inverses for our transcendental functions, and prove a large collection of lemmas about the functions defined.

In Chapter 6 we describe how we have used two different methods to provide automation for checking continuity of functions in PVS. In general the question of continuity is undecidable, but using restrictions we can still prove continuity of a very useful collection of functions. The first method is based purely on a new strategy which inspects the proof goal and tries to determine which theorems to apply. The second method relies on type judgements in the PVS theories to enable matching and then uses the built-in strategy `grind` to do the actual search for a proof. This second method turns out to be surprisingly powerful, although some problems still exist as the language for type judgement is somewhat restricted. We also show that this second method is more generally applicable with only minor changes by using it to check automatically for existence of limits. Finally, we discuss the issues of calculating values within PVS and of functions defined by cases.

Appendix A contains all our strategies and Appendix B contains examples files used for testing the strategies and shows the test results. Appendix C contains the `README` file from our library, giving an overview of the origin of the files in the library to distinguish between Dutertre's files and our own. Finally, we have included a CD-ROM with a copy of the complete library.

Chapter 2

Background and Motivation

In this chapter we give the main motivation for our work, namely to support Computer Algebra Systems (CASs) using theorem proving. We begin by giving an overview of CASs and some related issues of correctness (Section 2.1). In Section 2.2 we give an introduction to higher order theorem proving and Section 2.3 describes some projects on formalising mathematics, with particular emphasis on reasoning about real numbers and real analysis. Finally in Section 2.4 we discuss various approaches to combining computer algebra systems and theorem proving and outline two applications, a look-up table for definite integrals (DITLU) and a prototype interface between the CAS Maple and the theorem prover PVS, both of which use our PVS library.

2.1 Computer Algebra Systems

Computer Algebra Systems (CASs) are widely used both in industry (by engineers) and in academia (by researchers and students alike). The systems can do many forms of mathematics, ranging from simple calculations through formulae manipulation to equation solving. The simplest use is comparable with that of using a beginner's calculator yet the systems are powerful enough to do substantial scientific calculations without becoming too difficult to use. The fact that CASs are fairly easy to use is one of the main reasons why they are so popular. Some well-known systems are axiom, Maple and Mathematica.

In this section we first briefly describe some areas of algebra of particular relevance to

CASs and how these differ from related areas in analysis. Then we look at three CASs, and finally we discuss correctness of CASs and give some examples of causes for incorrect calculations in CASs.

2.1.1 Computer Algebra

Computer algebra systems are designed to do computation in areas such as algebra, factorisation of polynomials, Gröbner bases, symbolic integration and solving differential equations. We will contrast this with true real analysis.

In computer algebra we work with a ring $\langle S, +, \cdot \rangle$ of *rational functions*, that is quotients of polynomials over the reals $\mathbf{R}$ extended with finitely many constants. For example,

$$f(x) = \frac{ax^3 + 5x}{cx + 1}$$

is a rational function.

We then extend this ring by adding a differentiation operator $D : S \to S$, such that for constants a

$$D(a) = 0 \tag{2.1}$$

and for variables x

$$D(x) = 1 \tag{2.2}$$

and $\forall f, g \in S$

$$D(f + g) = D(f) + D(g) \text{ and} \tag{2.3}$$
$$D(fg) = D(f) \cdot g + f \cdot D(g) \tag{2.4}$$

Note that this implies that for all $r \in \mathbf{R}$, $D(r) = 0$, and that

$$D\left(\frac{f}{g}\right) = \frac{D(f) \cdot g - f \cdot D(g)}{g^2} \tag{2.5}$$

since

$$D\left(\frac{f}{g} \cdot g\right) = D(f) \tag{2.6}$$

by definition of division and

$$D\left(\frac{f}{g}\cdot g\right) = D\left(\frac{f}{g}\right)\cdot g + D(g)\cdot\frac{f}{g} \tag{2.7}$$

by Equation (2.4).

The extended ring is then a *differential ring*. In differential rings there are no notions of limits, continuity, areas under curves or the like. Also *rational functions* are not regarded as functions from $\mathbf{R}$ to $\mathbf{R}$, but simply as quotients of polynomials.

The *anti-derivative* of $f \in S$ is then some $g \in S$ such that $D(g) = f$. With the definitions of the differentiation operator D above, not all elements of S have an anti-derivative, for example $\frac{1}{x}$ does not. However, we can extend S and D with an operator ln such that $\ln(x)$ is the derivative of $\frac{1}{x}$. We also introduce the *elementary functions*. These are based on the rational functions and *transcendental functions*, such as exp, cos and sin. By combining rational and transcendental functions we get elementary functions. For example,

$$f(x) = \frac{\exp(ax^3 + \cos(3x))}{\cos(cx+1)}$$

is an elementary function.

What has now become known as *the Risch Algorithm* [GCL92, Bro98] decides if an elementary function has an anti-derivative expressible by elementary functions, and if so, calculates it. The Risch Algorithm is used in most CASs and other systems implementing integration (Section 2.1.3). It relies on purely algebraic properties and procedures such as irreducibility and square-free factorisation.

In the context of CASs, we might define *definite integration* by

$$\int_a^b f(x)\,dx = g(a) - g(b)$$

where g is an anti-derivative of f. However, as we shall see in Chapter 3 this does not correspond to the usual analytic definition, and gives different answers in some cases.

Analysis

In *analysis* we work with values of functions, and *rational functions* are functions from subsets of $\mathbf{R}$ to $\mathbf{R}$ based on constants, identity, addition, subtraction, multiplication and

division. For example, if $c \neq 0$ then

$$f(x) = \frac{ax^3 + 5x}{cx + 1}$$

defined on $\mathbf{R} \setminus \left\{-\frac{1}{c}\right\}$ is a rational function in analysis, if $c = 0$ then the function is defined on all of $\mathbf{R}$. We might use the classical ε–δ definitions to define limits of functions, continuity and differentiability.

The notion of integration within analysis is about *areas under curves*, and we calculate definite integrals via the Fundamental Theorem of Calculus:

Theorem 2.1 (The Fundamental Theorem of Calculus) *Let f be continuous on a closed interval $[a, b]$ in $\mathbf{R}$, let F be continuous on $[a, b]$ and differentiable on the open interval (a, b) with $\forall x \in (a, b)\,.\, F'(x) = f(x)$. Then*

$$\int_a^b f(x)\, dx = F(b) - F(a)$$

In simple cases, the CAS definition of definite integration corresponds to the analytic one, for example for

$$\int_a^b 4x + 2\, dx = 2b^2 + 2b - 2a^2 - 2a$$

However, for more complicated cases, the two definitions diverge. As we shall see in Chapter 3 this causes CASs often to give unexpected results when doing definite integration.

2.1.2 Systems

There are several different CASs with different emphasis on issues such as correctness/safety and usability. In this section we briefly describe three of the main systems, namely axiom [JS92], Maple [CGG+92] and Mathematica [Wol91]. Common features for all three systems are that they provide facilities for interactive calculations as well as a programming language, thus enabling the users to develop new libraries. They also all have a large collection of built-in procedures for both symbolic and numerical calculations and manipulations of objects such as polynomials, integrals, differential equations and matrices.

axiom is strongly typed, and this is used to ensure sanity of the user input. For example, the axiom class `Matrix` implements matrices over any ring, whereas matrices over

say hash tables are not permitted. Likewise, axiom tries to ensure, using both typing and side conditions, that any method or transformation fits the given arguments. This becomes particularly apparent when using parameters, since axiom will try to prevent for instance definite integration of a function over a point where the function is undefined. Whereas this approach makes axiom much safer than most other CASs, it also makes it more difficult to use, since the level of precision about the context required to get results from axiom is often higher than that used when doing calculations by hand.

Maple also uses types, however not in any strict fashion. Procedures may accept input of a different form from what may be expected from the documentation, leading to unexpected results. Also, Maple does not in general attempt to ensure that a procedure applies to its input. Maple has a command `assume` which allows the user to state assumptions as in `assume(x>0)`. The intended semantics is that $x > 0$ is added to the current context, but this seems to be poorly implemented. As `assume` is a relatively new facility, many Maple commands have not been adapted to take these assumptions into account.

Mathematica also supports various datatypes, but as with Maple they are used fairly loosely. However, Mathematica perform more checks than Maple on its input and procedure application.

One might say that Maple appears to work in a fairly naive mode, trusting the user to give proper input, axiom then assumes that the user wants and/or needs the input checked and will be able to provide further information, should axiom need it to perform the calculation. Mathematica sits somewhere between these two extremes. Maple and Mathematica are certainly easier to learn initially, but due to their looser typing and lack of checks, they also give less reliable results, although they give results in more circumstances.

2.1.3 Correctness of CASs

Obviously, the user would like the answers from the CAS to be *correct*. However, the notion of correctness as understood by the user might depend on the application. For instance, some equation might have two different solutions according to whether a certain value is negative or positive. An engineer might think it is obvious that the value is positive, as this

is always the case within his domain, and so might expect simply to get the one answer corresponding to that case. On the other hand a mathematician might wish to explore fully the solution to the equation and so would need both answers.

Different CASs use different approaches to this question, axiom will generally only give an answer if it is quite certain it is a correct one and it uses types to try and ensure this, whereas other systems will often give a result that might only hold under certain assumptions, although they will not indicate that. With a more complete system, the user will in general have to know more about the mathematics being used, in order to choose the right solution (or get one at all), but with systems which are easier to use, one might not get full solutions. Thus there is a tradeoff between correctness and usability.

Why does incorrectness occur?

In the following sections we shall see examples of how CASs give unexpected and sometimes wrong results, but let us first consider how these errors might be explained. There are at least six different origins of errors within CASs:

Simple Bugs Whereas a programming error might not be easy to detect or correct, it is simple in the sense that it just a programming error, rather than reflecting problems with combining the underlying mathematics, that is algebra and analysis.

Side Conditions Many mathematical theorems have preconditions that must be checked to ensure that the theorem applies. CASs do not in general have mechanisms in place to handle preconditions such as these, and a design decision was made largely to ignore them. A particularly obvious example of this error is doing definite integration via the fundamental theorem of calculus (Section 2.1.3), where the integrand must be continuous for the theorem to apply.

No "Good" Solution Not all problems have nice solutions. Some are undecidable (like "is f continuous?"), and some questions do not have well-defined answers (for example elementary functions do not have recognised normal forms – should one write $\cos(x)\sin(x)$ or $\frac{1}{2}\sin(2x)$).

Parameters By parameters we mean constants whose values are not (yet) know. Many subject areas within calculations can be handled by numerical methods, however for

problems containing parameters these methods do not apply directly, although some experimentation might be possible. Also, many algorithms in computer algebra only apply to problems without parameters. For instance, the usual implementations of the Risch algorithm for finding anti-derivatives do not handle parameters well.

Semantics CASs do not have formal written semantics and the documentation is not usually very precise, so the only way to decide on the usage of a command is by guessing based on the documentation and experiments. Also, it is not always clear what has been implemented (for instance when there are several variants of the same algorithm), what modifications were made to a "standard" algorithm, and what additional results are provided in look-up tables.

Algebra versus Analysis As discussed in Section 2.1.1 the notion of functions, differentiability and integration differs in the areas of algebra and analysis. This will lead to inconsistencies between users' points of view and their expectations, and the results returned by the CAS.

Combining the above problem areas gives the potential for many more or less obvious difficulties for CASs, however there is one more major source of problems: imprecise notation. In general, CAS users (like most mathematicians and engineers) take context for granted, for instance in an analysis textbook, one does not continue to state that $\frac{1}{x}$ is a function defined on say $\mathbf{R} \setminus \{0\}$. However, by not being precise there is a great risk of misinterpretation. Take for example the expression $\frac{1}{x} + 1 - \frac{1}{x}$; it is obviously constant 1, but is it defined at 0 or not? Algebraically, it is the polynomial which is constant one, but analytically it is a constant function from $\mathbf{R} \setminus \{0\}$ with the value one. So in analysis there is an implicit assumption that $x \neq 0$ when we reason about the above expression.

We will now take a closer look at three areas (simplification, continuity and symbolic definite integration) where some of these problems arise. We will use examples to illustrate situations in which a user might be not just confused but even led to perform incorrect calculations.

Simplification

When we are doing calculations (whether within CASs or not) we wish to present the result in a suitable form. However, for arithmetic expressions there is no such thing as

a canonical form, for example the polynomial $x^3 - 2x^2 - 5x + 6$ can also be written as $(x-3)(x+2)(x-1)$, and depending on the context one form might be more appropriate than the other. If we are trying to match the polynomial to a look-up table (which is often used within CASs) we might prefer the first form, dependent on the table of course. But if we are looking for roots of the polynomial, the second form is more useful as it tells us straight away that the roots are -2, 1 and 3. So the context will influence our choice of how to represent an answer.

Particularly when working with trigonometric functions, CASs often manipulate the input to obtain some different internal representation, for instance in Maple, $\sin^2(x)$ is represented by $1 - \cos^2(x)$. Sometimes this causes the result to be displayed in terms of the internal representation, which might then look nothing like the user's input. Also, determining when a certain transformation applies is not trivial, for example axiom relies on the transformation

$$2\arctan(x) = \arctan\left(\frac{2x}{1-x^2}\right) \tag{2.8}$$

internally regardless of the possible values of x (see Section 3.1). However, as the following argument shows, this is false.

We know that

$$\forall x_1 \in \mathbf{R}_- \,.\, \arctan(x_1) < 0$$
$$\forall x_2 \in \mathbf{R}_+ \,.\, \arctan(x_2) > 0$$
$$\arctan(0) = 0$$

Now for $x = 3$, we see that $2\arctan(x) > 0$ and $\arctan\left(\frac{2x}{1-x^2}\right) < 0$, thus for $x = 3$ (2.8) is not true.

In fact, for all x such that x and $\frac{2x}{1-x^2}$ have opposite signs (2.8) is false, thus the transformation only applies to $x \in (-1, 1)$. So axiom is relying on a transformation which is not true everywhere. Since axiom is being phased out, correcting this problem is not considered a priority neither by developers nor by users.

Continuity

CASs work on differential rings, so there is no notion of continuity. However when working with CASs, for example in modelling, one might be interested in certain analytic properties

of a function, such as whether it is continuous on some or all of its domain. Whereas CASs were not originally intended to work on analysis, most systems now provide some support for these issues.

In Maple we can use the command `iscont` to check if a function is continuous. It is called with `iscont(f(x),x=a..b)` where `a..b` is a range which may contain $-\infty$ and/or ∞. By default the range is considered to be open. The answer returned is one of `true` (if the function is continuous in the interval), `false` (if it is not) or `FAIL` (if Maple can not determine one way or the other). The Maple documentation does not tell us what methods `iscont` is based on, however it would not be using the actual analytic ε–δ definition of continuity, since this is really outside the capabilities of Maple and would need actual reasoning.

Even though `iscont` does handle very simple functions well the following example shows that it is not trustworthy.

Example 2.1 *Consider the function* $f(x) = \frac{1}{2+\cos(x)}$ *which is defined on the reals and continuous everywhere, since* $\forall x \in \mathbf{R} \,.\, 2 + \cos(x) \neq 0$.

```
> iscont(1/(2+cos(x)),x=-infinity..infinity);
```

false

So `iscont` *returns the wrong answer.* □

Even though Maple has support for "checking continuity of functions", it is clearly not very reliable. We have not done extensive testing of `iscont`, and we have not detected an example where Maple gives an incorrect positive answer, that is returns true when the function is *not* continuous. In Section 2.4.3 we shall present a way of improving this feature of Maple.

Integration

Another area in which CASs in general do quite badly is definite symbolic integration, for example

$$\int_a^b \frac{1}{x+c}\,dx$$

Closely related to this problem are those of indefinite integration, such as

$$\int \frac{1}{x+c}\,dx$$

and of definite integration without parameters, such as

$$\int_{1.2}^{\pi} \frac{1}{x+\exp(3.5)}\,dx$$

Indefinite integration may be done either numerically or symbolically. In the case of symbolic indefinite integration, the standard method used is what is known as the Risch Algorithm. Definite integration can be done either by using indefinite integration and the fundamental theorem of calculus, or by using numerical methods. However, when there are parameters in either the limits or the integrand, numerical methods can not be used to find a parameterised answer, although one can experiment with values of the parameters.

In Chapter 3 we will give a case study of how the systems axiom, Maple, Mathematica and Matlab deals with definite integration with parameters, but let us now consider some standard methods mentioned above.

Indefinite Integration In 1970 Risch described an algorithm which takes an elementary function and either gives an anti-derivative which is also expressed in elementary functions over that function or proves that there is no such integral [GCL92, Bro98]. The algorithm has become known as the *Risch Algorithm*, and over the years refinements and improvements of it have taken place. It is now used in most CASs and other systems implementing integration. When the integrand (the f in $\int f\,dx$) contains parameters the standard Risch algorithm does not apply. Bronstein has developed methods to handle some of these cases and presents algorithms for these in [Bro96], although no implementations are currently available.

Numerical Definite Integration There are several different standard methods for numerical integration, the two main categories are Newton-Cotes Integration and Gaussian Quadratures. Both groups of methods work by approximating the integral. This is done by partitioning the interval and then using rectangles or more complicated shapes to approximate the area. The Gaussian Quadratures use more sophisticated shapes, however this method can only be used if the integrand is smooth, and often, in the case of numerical integration, the data is measured and so not known to

be smooth. Newton-Cotes applies in this case and with the computing power available today, it is not a problem that Newton-Cotes is slightly less efficient than using Gaussian Quadratures.

Symbolic Definite Integration The fundamental theorem of calculus (Theorem 2.1) plays an important role in symbolic definite integration. Using this theorem, symbolic definite integration is reduced to indefinite integration (which using the Risch Algorithm is considered a solved problem) and to checking the prerequisites of the theorem. However, CASs in general do not have internal support to deal with side conditions such as "f is continuous", so it is hardly surprising that CASs get even fairly simple definite integrals wrong.

2.2 Theorem Proving

Theorem proving is used extensively in verification of both software and hardware. Traditionally, extensive testing is used to show that an implementation or a piece of hardware obeys its specification, although due to the number of states it is usually not possible to cover all cases. Formal verification is then being used to reason about a model of the implementation, so that given a specification for the implementation or hardware, we can attempt to prove that it has the desired behaviour. One major issue here is that of ensuring that the model accurately represents the implementation or hardware we are verifying.

2.2.1 Interactive versus Automated Theorem Proving

It would of course be desirable to have totally automatic theorem proving, however even simple problems of first order logic can be undecidable. There are essentially three different approaches to dealing with the issue of automation: One might restrict the logic used in the system and thus provide powerful automation; one might use *proof checking* where the user constructs the proof and guides the theorem prover in checking it, perhaps filling in small steps during the proof; and finally one might use the approach introduced by Edinburgh LCF project [GMW79], namely have a fixed set of low-level automatic commands and a language for the user to write new application specific strategies based on this set of commands. The low-level commands are usually referred to as *tactics* and the composed

commands as *tacticals*.

Dependent on the application area we might prefer any of these methods. If a restricted logic is enough to express and reason about our problems, then we would probably prefer the high level of automation as that obviously would be an easier system to use, however, if this is not the case we might wish to write our own strategies in order to provide automation for our particular application. Some areas consist of collections of proofs which are not too similar, and in these cases the effort of developing new strategies would probably not correspond to the gain of having them, and so we could use a proof checker with less support for automation.

2.2.2 Systems: Higher Order Theorem Provers

There are several different first order and higher order theorem provers, in this section we will describe the basics of four of the higher order theorem provers, namely Coq [Tea], HOL [GM93], Isabelle [Pau93] and PVS [OSRSC99b].

Common to these system are that they provide some level of automation, but, more importantly, all have a strategy language enabling users to develop and apply their own strategies and thus build up application specific automation. HOL, Isabelle and Coq are all LCF-style theorem provers, that is they have a small core of proof commands which can be fairly easily verified. This core is then used to construct higher level commands. Since the core is small for an LCF-style theorem prover, the risk of inconsistencies in the theorem prover is less than in other theorem provers.

PVS is based on classical higher order logic and is strongly typed and supports sub-typing. It is implemented in Lisp, which also forms the basis for PVS's strategy language. PVS contains many predefined definitions and theorems supporting a variety of application areas. These are organised into *theories*, which may contain axioms, assumptions and theorems – theorems may be proven or unproven, PVS tells the user if a proof relies on an unproven theorem.

HOL is also based on classical higher order logic and is typed, however HOL uses constants in expressions to infer types of variables, so a user might not be aware of the type HOL assigned a variable. HOL is essentially command-line based, so the user

is responsible for "book-keeping" to keep track of proof steps. However, several interfaces are now available. HOL theories contain only proven theorems. HOL is implemented in ML and also uses ML for its strategy language.

Isabelle is a generic logical framework which can be instantiated with various logics. There are some well-developed logics within Isabelle already, amongst these are: Isabelle/HOL (classical higher order logic), Isabelle/HOLCF (higher order logic with computable functions) and Isabelle/ZF (first order classical and intuitionistic logic with set theory). Isabelle/HOL has the most applications already available, covering areas such as mathematical analysis and verification of a subset of Java. As HOL, Isabelle is implemented in ML

Coq is based on calculus of inductive constructions, a typed λ-calculus. Coq is implemented in Objective Caml, which is a dialect of ML.

2.3 Formal Mathematics

By *formal mathematics* we mean mathematics done using computer programs based on some sort of logical reasoning, thereby assuring that the user can not do incorrect mathematics. This assurance is of course dependent on the correctness of the computer program itself.

In this section we shall briefly describe various types of computer programs which support formal mathematics. Section 2.3.2 is about higher order theorem proving for real numbers with support for real analysis, whereas in Section 2.3.1 we mention some other projects working on formalising mathematics. Section 2.3.3 describes two applications of PVS and ACL2 using real numbers.

2.3.1 Projects on Formalising Mathematics

Over the years there have been several projects dealing with formalising parts of mathematics. The oldest and possibly best known are Automath and Mizar.

The Automath project [Rez83] originated at Eindhoven University of Technology in 1967.

The idea is to have a powerful language in which one can state mathematical definitions and theorems, however only mathematics which is actually true will correspond to correct sentences in the language. As a means of getting experience with large amounts of formalisation E. Landau's book "Grundlagen der Analysis" was translated successfully into Automath. Automath is a proof checker.

Mizar [Rud92] was started in the early 70's by Trybulec at Plock Scientific Society, Poland. The purpose is to have a tool in which theorems can be described in text form and then checked for correctness. However an interesting addition is that there are also tools which allow the theorems along with their proofs to be formatted graphically.

MathPert [Bee89] is developed by Beeson. It is intended as a tool for learning mathematics, particularly algebra, trigonometry and analysis. The user poses a question and can then attempt to answer it using the facilities of MathPert. As MathPert keeps track of the problem and partial solution, it is not possible to make simple typing errors, like changing the sign of a number. MathPert also has a facility which allows the user to step through MathPert's suggested solution to a problem. Closely linked to MathPert is the program Weierstrass [Bee98] which does automatic ε–δ proofs of continuity. Weierstrass uses calculations from MathPert and is meant to be used to produce proofs of "peer-review" standard, that is proofs to be read and checked by humans, rather than be trusted blindly. Since both MathPert and Weierstrass have been developed to be used for guidance rather than to be trusted as for example a theorem prover, they have not been formally verified.

2.3.2 Higher Order Theorem Proving for Real Analysis

There is currently considerable interest in real analysis among users and developers of higher order theorem provers which is made clear by the range of implementations and applications available and the number of ongoing projects. In this section we will describe some implementations and give two examples of applications for formal mathematics.

Basic Real Analysis in PVS

The real numbers are built into PVS, they are axiomatised with part of the knowledge about them stored in the decision procedures, see Section 4.2. PVS also supports various

subtypes of the real numbers, such as positive reals, nonzero reals, rationals, integers and naturals. Furthermore, PVS provides the usual operations on numbers and a large collection of theorems about (in)equalities of real-valued expressions.

Dutertre [Dut96] implemented a real analysis library in PVS. He uses classical analysis based on ε–δ definitions to define and reason about limits, continuity and differentiation of real-valued functions, see Section 4.3. The definitions of limits of functions contained an error, making the definitions of existence of a limit and of continuity equivalent in many cases. That, of course, is not correct, and we identified and rectified this error.

An axiomatisation of trigonometric functions (cosine, sine, tangent and their inverses) has been done in PVS by researchers at ICASE and NASA Langley [MCB+]. In Section 5.5 we will compare this to our own PVS library.

More Advanced Real Analysis in HOL Light

Harrison [Har98] developed support for real analysis in HOL Light, his own version of HOL (see Section 5.1.1). The reals are constructed using a variation of Cantor's method, and he uses convergence nets rather than the ε–δ definitions used in Dutertre's PVS implementation. Harrison also takes his library further, as he develops support for transcendental functions, such as exp, cos, sin and their inverses. He also defines and proves various theorems about Kurzweil-Henstock gauge integration. The Kurzweil-Henstock gauge integral is a generalisation of the usual Riemann integral and Lebesgue integral, and a function f has a Lebesgue integral exactly if both f and $|f|$ have Kurzweil-Henstock gauge integrals.

Nonstandard Analysis in Isabelle

Fleuriot has implemented a real analysis library in Isabelle based on nonstandard analysis [Fle00], linking definitions and theorems in nonstandard analysis to those in classical analysis. In nonstandard analysis we have infinitesimals, that is numbers x such that $\forall r \in \mathbf{R}_+ \,.\, |x| < r$. This allows many of the properties of interest in analysis, such as uniform continuity, to be expressed in a much simpler way than in classical analysis. In classical analysis, uniform continuity of a function f at a point x is defined by

$$\forall \varepsilon > 0 \, \exists \delta > 0 \, \forall y \,.\, |x - y| < \delta \Rightarrow |f(x) - f(y)| < \varepsilon$$

Given the *infinitely close* relation $\approx$ defined by

$$x \approx y \equiv x - y \text{ is an infinitesimal}$$

we can define uniform continuity in nonstandard analysis by

$$x \approx y \Rightarrow f^*(x) \approx f^*(y)$$

where f^* is the extension of f from the real numbers to the hyperreals [Fle00].

Using nonstandard analysis, Fleuriot found that not only many definitions as the one above but also theorems and more importantly proofs were simpler than when using classical analysis. However, to make it obvious that the nonstandard analysis definitions and theorems do indeed correspond to the better known ones of classical analysis, he also proved equivalences between definitions and theorems in the two different notions along the way, thus adding to the proof burden.

Axiomatisation of the Real Numbers in Coq

Mayero developed a classical axiomatisation of the real numbers in Coq [May99] in order to prove the Three Gap Theorem, which says that points distributed on a circle by an angle α divide the circle into gaps of at most three different lengths. She declares the type of reals and the constructors zero, one, addition, multiplication, unary minus, inversion (as in $\frac{1}{x}$), and strictly-less-than. From this foundation she defines greater-than, less-than-or-equal-to, greater-than-or-equal-to, and binary minus. The usual properties of addition and multiplication, such as commutative, associative, and distributive are axiomatised. Following this is a collection of lemmas about (in)equalities on real-valued expressions. This is similar to, but not as extensive as, that in PVS.

In [GPWZ00] is an overview of a proof of the Fundamental Theorem of Algebra, this is done without an actual implementation of the real numbers but using a set of axioms describing the reals. Since they need the constructive reals rather than the classical reals, they could not use any existing implementations. However, [CG00] describes an implementation of constructive reals using streams of the numbers $\{-1, 0, 1\}$, similar to representations used in exact real arithmetic. In order to justify the implementation as a proper construction of the reals, they simplify the axiomatisation in [GPWZ00] and show that their implementation satisfies these axioms.

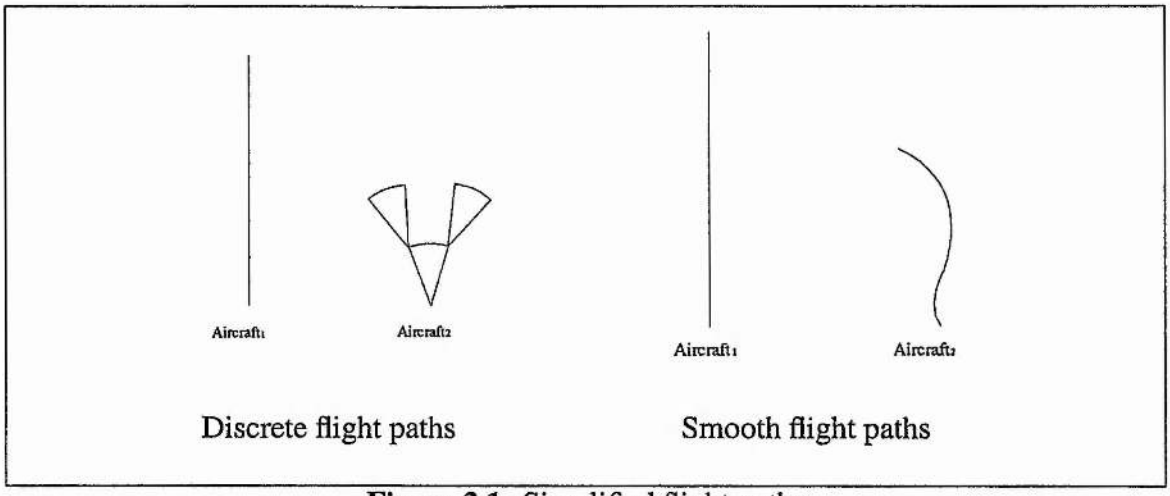


Figure 2.1: Simplified flight paths

2.3.3 Applications

There are many applications of the various formalisations of mathematics. In this section we will briefly describe just two projects, one about software verification and one about hardware verification.

AILS

Carreño and Muñoz [CM00] used a combination of PVS with the ICASE/NASA Langley trigonometric library [MCB+] and the CAS Mathematica to verify an alerting algorithm for aircrafts landing on parallel runways. Initially, the flight paths were considered to be discrete in that they are modelled as 3° circle segments, as shown in Figure 2.1. They proved that a collision warning would occur at least 9 seconds before a possible collision. However, by describing the flight paths using continuous mathematics, also shown in Figure 2.1, they could prove that a collision warning would occur at least 10 seconds before a collision. So in this case using continuous mathematics made a big difference to the end result.

Floating Point Arithmetic

Russinoff [Rus98] used ACL2 [KMM00] to verify the floating point multiplication, division and square root algorithms of the AMD-K7 microprocessor against IEEE Standard

754 for binary floating-point arithmetic. These algorithms are implemented directly in hardware, and Russinoff models the hardware implementations at register level where the operations are logical functions on bit vectors. However, the specifications for the algorithms are best expressed in terms of mathematics, and so the verification is exactly showing that the low-level model satisfies the higher level specification. Russinoff found two design flaws in the division algorithm although it had been tested on about 80 million test vectors beforehand. He also found that the multiplier could be reduced in width from 76 to 75 bits, even though it was believed to be minimal and thereby most efficient.

Harrison [Har00] used HOL Light to verify floating point division algorithms for the IA-64 computer architecture. IA-64 directly supports *floating point reciprocal approximation*, that is floating point approximation of $\frac{1}{x}$. However general floating point division is implemented in software rather than in hardware. Intel then provides a collection of division algorithms for use by IA-64 developers.

2.4 Systems

In this section we first describe three different approaches to combining CASs and theorem provers (TPs) and then we describe two applications developed by the CAAR group in St Andrews.

2.4.1 Combining CASs and TPs

Theorem provers and computer algebra systems have different strong points. CASs in general are very good at solving equations and for example finding roots of a polynomial, whereas most theorem provers can quite easily prove that a root is indeed a root, but are not usually able to calculate the roots in the first place. However, TPs naturally handle theorems with side conditions well, and as many theorems of mathematics either have side conditions or even hidden assumptions theorem proving lends itself well to mathematics.

Apart from the usual problems when combining systems in actually sending and receiving data, we also need to consider the semantics of the data being passed from one system to the other. Except for a few cases, such as Analytica [BCZ98] which is built inside

Mathematica, the two systems do not normally share their syntax, let alone their semantics. OpenMath [Dew00] is one project dealing with this issue. For each system one can write a *phrase book* which contains information on how expressions are to be interpreted. This has been successfully used with at least the following systems: axiom, Mathematica and GAP. If one has two specific systems linked up, such as with our Maple-PVS interface (see Section 2.4.3), one can write a translator for just those two systems. However, that might be too restrictive for further development of either system, as the translator would be dependent on both systems rather that just one of them.

There has been quite a lot of research into how we might exploit the best features of both CASs and TPs, and there are essentially three different ways of combining CASs and TPs. Each of these have different strong points, and so appeal to different user groups.

(i) CASs supporting TPs With a CAS supporting a TP, the TP will query the CAS, for example for solutions to an equation or an indefinite integral. However, in general we can not trust a CAS to the same extent we would like to be able to trust a TP, so the TP might decide only to use the result from the CAS if it can verify that it is correct.

For example, if the TP asked the CAS to find the roots of a polynomial, the TP might wish to check that the answers from the CAS are indeed roots of the polynomial. Of course, the TP can only do this, if it knows enough arithmetic to calculate sums, products and integer powers, but most TPs do. However for more complicated CAS operations, proving correctness of the result in the TP might require development of large theories in the TP and even then be difficult.

Using a CAS from within a TP would definitely enhance the TP, and provided the mechanisms are there to check the answer it could be quite easy to use for TP user.

Examples using this approach includes interfaces between the following systems: Maple-HOL [HT93], Maple-Isabelle [BHC95] and Weyl-NuPrl [Jac95]. Also, the proof development tool Omega [KKS98] uses a combination of Maple and Gap to do computations.

A different approach is used with Theorema [BJK+97] and Analytica [BCZ98], which are both built within the CAS Mathematica. This of course means that they are totally reliant on Mathematica and so more prone to suffer from any errors in the CAS than the systems which use a separate prover.

(ii) TPs supporting CASs With a TP supporting a CAS, the main benefit is that the CAS user might gain more certainty in the results and that the TP can handle side conditions, so that we can work with areas of mathematics that require this.

The CAS might use the TP to prove that for example the roots of a polynomial are just that. Comparing this approach to that of CAS supporting TP, we see that here the less trusted system (the CAS) is calling the more trusted system (the TP), and so there is no need for checking the result from the other system.

Using a TP from inside a CAS adds a lot to the complication of using the CAS, as most CASs are in general fairly accessible, whereas that is not true of TPs. So for this approach to work, the TP would need to be almost "hidden" from the CAS user, so that if the CAS user is not a TP user, he can still use the CAS and gain the benefits from the combined system.

An example using this approach is our Maple-PVS interface, see Section 2.4.3.

(iii) CASs and TPs side-by-side The third option is to have the CAS and the TP running side-by-side with some kind of "broker" in between them. The broker might control whether a query goes to one system or the other, and during say a calculation in the CAS, the CAS might ask the TP (via the broker) to prove some particular property.

Using this approach would enable scenarios from both of the methods mentioned above, however controlling which system does what and avoiding circular queries between the two systems might not be easy. One could use OpenMath [Dew00] as a standard for the data to be exchanged.

2.4.2 DITLU

We have successfully used our PVS library of elementary functions and real analysis with DITLU [AGLM99], a look-up table for symbolic definite integrals, which CASs do not in general handle very well (see Chapter 3). The table works on integrals of the form

$$\int_a^b f(x)dx.$$

where both the limits and the function may include parameters. Dependent on the function and the values (or ranges) of the parameters the integral might be undefined or take a particular value. In the table this is represented as case-statements and we used PVS to

eliminate the cases that could definitely not occur for a given query. For example, one of the entries in the table is

$$\int_b^c \frac{1}{p+qx}dx = \begin{cases} 0 & \text{if } (b=c) \\ \hline undefined & \text{if } (q \neq 0) \wedge (b \neq c) \wedge \\ & ((b=-\frac{p}{q}) \vee (c=-\frac{p}{q})) \\ \hline \dfrac{\ln|qc+p| - \ln|qb+p|}{q} & \text{if } (q \neq 0) \wedge (b \neq c) \wedge \\ & (b \neq -\frac{p}{q}) \wedge (c \neq -\frac{p}{q})) \\ \hline \dfrac{c-b}{p} & \text{if } (b \neq c) \wedge (p \neq 0) \wedge (q=0) \\ \hline undefined & \text{if } (b \neq c) \wedge (p=0) \wedge (q=0) \end{cases}$$

So with a query such as

$$\int_{-a}^{a+1} \frac{1}{x}dx.$$

we get the following match with the entry above:

$$b=-a,\ c=a+1,\ p=0,\ q=1$$

We see that dependent on the value of a the first three cases might occur, case 1 if $a=\frac{-1}{2}$ and cases 2 and 3 if $a=-1$. However, the last two cases will not occur with this query, since $q=1$. So the result of the query is the first three cases only. We used PVS to check these side conditions in order to remove the cases which can not occur.

2.4.3 Maple-PVS

In our research group we have developed an interface between the CAS Maple (version 6) and the TP PVS, so that Maple users may benefit from the added confidence in their results by having them or side conditions checked by PVS. The actual interface is based on Maple's interface to external C functions through which we provide a low-level interface to PVS, with Maple running PVS as a subprocess.

The Maple user can then start PVS, query it and inspect the answers and stop PVS. We use a Tcl/Tk window to display output from PVS and allow interaction with PVS.

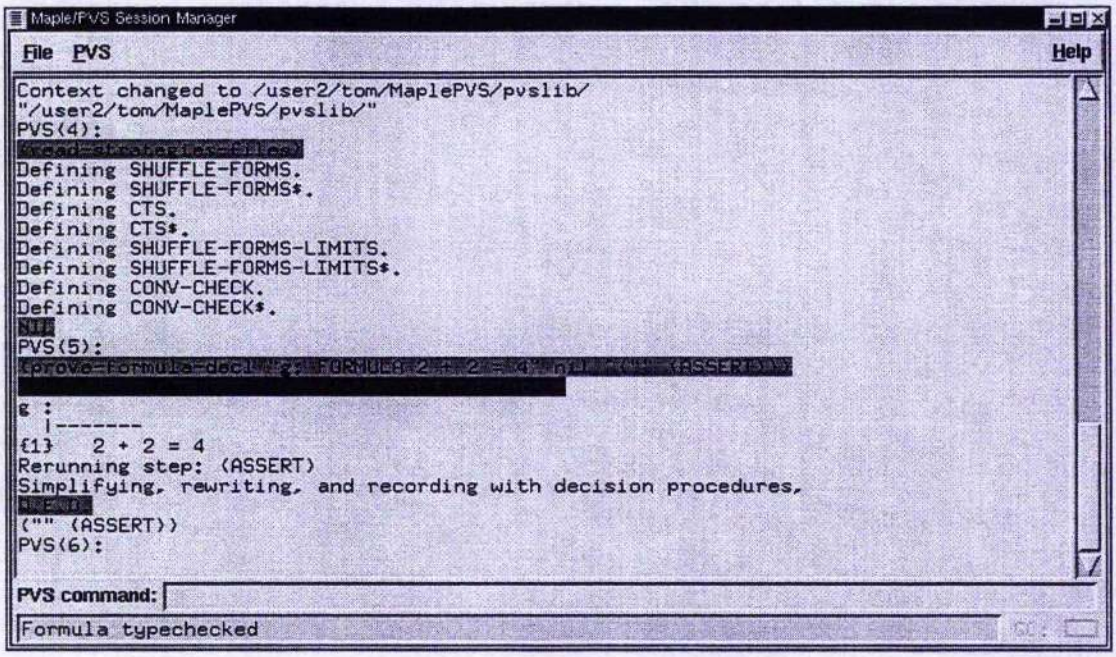


Figure 2.2: Tcl/Tk window for the Maple-PVS interface

Example 2.2 (A Simple Example) *To illustrate the basic use of the interface, let us prove* $2 + 2 = 4$. *First we start up PVS:*

```
> pvs := PvsStart("../pvslib"):
```

where "`../pvslib`" *gives the context to be used by PVS. This starts up PVS and opens communication and displays the Tcl/Tk window. We can then query PVS:*

```
> ex1 := PvsProve(pvs, "g:  FORMULA 2 + 2 = 4", "", ""):
```

This causes the PVS session identified by `pvs` *to try and prove the formula* `2 + 2 = 4` *using no library files (just the built-in theories of PVS) and the default proof command* `assert`. *The result of running this command in Maple is displayed in the Tcl/Tk window as shown in Figure 2.2.* `PvsProve` *returns various information about the proof attempt, so in order to check if the proof command proved the formula, we run the following:*

```
> PvsQEDfind(ex1);
```

true

which checks if a QED occurred in the output from PVS to signal that the proof was successful. □

This example shows how we can start up PVS and use it to prove a simple statement. However, it is only when dealing with more complicated mathematics that we gain much from using PVS as well, as we would expect Maple to correctly calculate $2 + 2 = 4$ itself. Let us now consider a more interesting example, namely that of determining if a function is continuous.

Example 2.3 (Safe Implementation of `iscont`) *Maple has the command* `iscont` *which checks if a function is continuous or not, however as shown in Example 2.1 it is not reliable.*

We can use our interface to implement a safer `PVSiscont`*, using the analysis library described in Chapter 5 and the strategies in Chapter 6 (see Figure 2.3).*

`PVSiscont` *takes three arguments: a reference to a running PVS process, a function and a range to check continuity over. First the function is converted to a string. We do not yet perform any translation from Maple syntax to PVS syntax but rather rely on the fact that the common functions are called the same in Maple and PVS. This also means that we are not considering whether the semantics of the functions are the same in the two systems.*

Next the left and right hand side of the range is extracted, the range might include infinity to either side of zero. This is handled in PVS by using `real` *if the range is* $-\infty..\infty$*, and* `posreal` *or* `negreal` *if the range is from zero to either infinity. Finally, if the range is, say,* $-6..\infty$ *we use the PVS type* `above(-6)`*, which consists of all reals greater than* -6*. Similarly for ranges from negative infinity to a number.*

Given the range we can compose the theorem to be proven in PVS. Due to the variations in ranges, we have six different cases, since the types must match in the theorem. Once the theorem is constructed, we first typecheck it, using the command `PvsTypecheck`*. If the constructed theorem does not typecheck in PVS there was an error in the input and* `PVSiscont` *terminates with an error. If the typechecking succeeds we attempt to prove the theorem using PVS. If this works,* `PVSiscont` *returns* true*, otherwise it returns* fail*, since absence of a proof in PVS does not mean that the theorem is false.* □

We do not have any support for finding and proving about discontinuities in PVS, but one could use Maple to try and calculate possible discontinuities, then try to verify this in PVS. That could be used to extend `PVSiscont` to a three valued function, just like Maple's own `iscont`. The current version of `PVSiscont` is only two valued, as it does not at all attempt to find discontinuities, however it is safer than `iscont` as we shall now see.

```
PVSiscont := proc(pvs, f::algebraic, arange::(name = range))
local rs, ls, thm, var, tctest, fs;
   fs := convert(f, string);
   var := lhs(arange);
   ls, rs := op(rhs(arange));
   if rs ≤ ls then error "Invalid range"
   elif ls = -∞ and rs = ∞ then
      thm := cat("forall(yyy:real): continuous(lambda(", var, ":real):",
                 fs, ",", "yyy)")
   elif ls = -∞ and rs = 0 then thm :=
      cat("forall(yyy:negreal): continuous(lambda(", var, ":negreal):",
           fs, ",", "yyy)")
   elif ls = 0 and rs = ∞ then thm :=
      cat("forall(yyy:posreal): continuous(lambda(", var, ":posreal):",
           fs, ",", "yyy)")
   elif ls = -∞ and not (rs = ∞) and not (rs = 0) then
      thm := cat("forall(yyy:below(", rs, ")): continuous(lambda(",
                 var, ":below(", rs, ")):", fs, ",","yyy)")
   elif rs = ∞ and not (ls = -∞) and not (ls = 0) then
      thm := cat("forall(yyy:above(", ls, ")): continuous(lambda(",
                 var, ":above(", ls, ")):", fs, ",", "yyy)")
   else thm := cat( "forall (yyy:I2(", ls, ",", rs,
      ")) : continuous(lambda (", var, ":I2(", ls, ",", rs, ")) : ",
      fs, ",", "yyy)" )
   end if;
   tctest := PvsTypecheck(pvs, thm, "top_analysis");
   if not PvsTCfind(tctest) then error "typecheck failure"
   elif PvsQEDfind(PvsProve(pvs, cat("g:FORMULA ", thm, ""),
                            "top_analysis", "cts")) then
      return true
   else return FAIL
   end if
end proc
```

Figure 2.3: The procedure `PVSiscont`

Example 2.4 (Example 2.1 Revisited) *In Example 2.1 we saw that the function* $f(x) = \frac{1}{2+\cos(x)}$ *is defined and continuous everywhere. We also saw that using Maple's command* `iscont` *gives the wrong answer in this case. But we can prove it in PVS using our new*

command PVSiscont*:*

```
> PVSiscont(pvs,1/(2+cos(x)),x=-infinity..infinity);
```

true

□

We can now prove a variety of functions continuous on different ranges:

```
> PVSiscont(pvs,1/(sin(x)+2),x=-infinity..infinity);
```

true

```
> PVSiscont(pvs,1/(exp(x)),x=0..infinity);
```

true

```
> PVSiscont(pvs,1/(sin(x)+2),x=-infinity..38);
```

true

```
> PVSiscont(pvs,x,x=6..infinity);
```

true

```
> PVSiscont(pvs,1/(exp(x)),x=-infinity..-1);
```

true

```
> PVSiscont(pvs, exp(-x),x=-infinity..0);
```

true

With procedures such as PVSiscont, the Maple user does not need to know any PVS to benefit from the interface. PVSiscont provides a safer version of iscont without any need for the user to interact directly with PVS.

We have also implemented a procedure PVSisdifferentiable which uses a PVS strategy deriv developed with Karen Petri to prove functions differentiable at particular points. Thus we can prove functions differentiable on various ranges:

```
> PVSisdifferentiable(pvs,1/(cos(x)+2),x=0..1);
```

true

```
> PVSisdifferentiable(pvs,1/(sin(x)+2),x=-infinity..infinity);
```

true

```
> PVSisdifferentiable(pvs,1/(exp(x)),x=0..infinity);
```

true

```
> PVSisdifferentiable(pvs,1/(sin(x)+2),x=-infinity..38);
```

true

```
> PVSisdifferentiable(pvs,x,x=-2..infinity);
```

true

```
> PVSisdifferentiable(pvs,1/(exp(x)),x=-infinity..-1);
```

true

```
> PVSisdifferentiable(pvs, exp(-x),x=-infinity..0);
```

true

Another application we have been experimenting with is that of solving differential equations. We simply use Maple's built-in differential equation solver, but perform certain checks on input and output. By checking that the input functions have certain properties, we can ensure for example that the differential equation has a (unique) solution. One of the properties we check is continuity, so `PVSiscont` would then be called by our differential equation solver.

2.5 Summary

One of the main reasons for errors in CASs is that they work with algebraic structures and do not check the analytic side conditions which are often necessary for results to be correct. One obvious example is that of definite integration, where the fundamental theorem of calculus (Theorem 2.1) applies only if certain analytic side conditions are met, yet CASs often use this theorem even though they have no mechanism to check the side conditions.

Also, many methods used in CASs do not generalise easily to cases including parameters in the input, and furthermore there is not usually clear well-defined semantics for CASs, making it difficult for a user to determine exactly which methods might apply to their problem, and which cases are covered by that method. We have seen examples of CASs (i) applying analytically wrong identities and (ii) giving obviously false results when checking a function for continuity.

Some of the CAS problems may be addressed using theorem proving, however it would need to be automatic theorem proving as opposed to interactive in order to shield the CAS user from the technicalities of theorem proving. There are already some implementations of

real numbers and real analysis in higher order theorem provers, such as basic real analysis in PVS, more advanced real analysis in HOL Light, nonstandard analysis in Isabelle and an axiomatisation of the real numbers in Coq. We have based our implementation on the real analysis library in PVS and extended it with transcendental functions and the required automation.

We have also seen two applications of formal continuous mathematics: AILS about verification of warning algorithms for airplanes landing on parallel runways and the use of an interface between Maple and PVS to improve certainty of Maple's results.

Chapter 3

Case Study: Integration

This chapter contains a case study of how four computer algebra systems (CASs), namely axiom, Maple, Mathematica and Matlab, perform when doing definite integration with parameters (Section 2.1.3).

The four CASs considered here are different in the degree of confidence we can have in their answers: axiom uses types to try and ensure calculations are correct and checks for side conditions (for example to avoid integrating through a pole), whereas Mathematica, Maple and Matlab seem to use more ad hoc methods for avoiding errors, with varying degree of success. They also quite clearly use tables for some results. Common to all the systems is that they do not inform the user of the assumptions they have made for the result to be valid. Due to the lack of precise documentation of the different systems, typically we can not determine the exact cause of (partially) erroneous results.

In Sections 3.1 to 3.4 we study definite integration in the four systems when the integrand involves arctan, $\frac{1}{x}$ and $\frac{1}{x^2+a}$. Within each section first each example is given along with its indefinite integral (as found in the standard table [Dwi57]) and definite integrals, these are called the *standard results*. Then the results of the tests are given and finally they are discussed. When listing the test results, we say that a result is the same as the standard result, if it can be brought to be identical using only associativity. Thus $a(b+c)$ is the same as $ca + ab$ whereas $2a - a2$ is not equal to 0. This way we may distinguish between the levels of *simplification* applied in the various examples.

When performing definite integration within analysis, which is what both axiom and Ma-

thematica claim to do, one must make sure that there are no poles in the interval of integration. For example, $f(x) = \frac{1}{x}$ has a pole at 0, so we can not use ordinary integration to compute an integral such as

$$\int_{-2}^{1} \frac{1}{x}\, dx \tag{3.1}$$

axiom is usually able to detect the possibility of a pole even when using parameters and then flags a warning to the user about this *potential pole*. When a pole occurs in the interval of integration we might attempt to calculate the *Cauchy principal value* (CPV) instead. The CPV of a divergent integral is the limit, if one exists, of the sum of the values of the integrals away from the singularity. For example, for the divergent integral in (3.1) the CPV is

$$\begin{aligned} CPV &= \lim_{r \to 0_+} \left(\int_{-2}^{-r} \frac{1}{x}\, dx + \int_{r}^{1} \frac{1}{x}\, dx \right) \\ &= \lim_{r \to 0_+} (\ln|-r| - \ln|-2| + \ln|1| - \ln|r|) \\ &= -\ln 2 \end{aligned}$$

Note that the limit is taken at the same rate at either side of the singularity. In Maple we can request an attempt to calculate a CPV for a definite integral such as the one above. This is however only implemented for non-parametric limits.

When the test results are listed, *sr* means that the result is the same as the standard result, *d* means a message was displayed saying that the integral is diverging (this is the same as axiom's "pole"), *n* means a naive answer (in each case this will be stated) and finally *pp* is used for axiom's "potential pole".

3.1 Variations over arctan

In this section examples of integrands featuring arctan are explored. Consider first

$$\int \arctan(x) dx$$

The *standard result* for the indefinite integral is then

$$\int \arctan(x) dx = x \arctan(x) - \frac{1}{2} \ln(1 + x^2) + C$$

Plots of both the integrand and the integral can be seen in Figure 3.1.

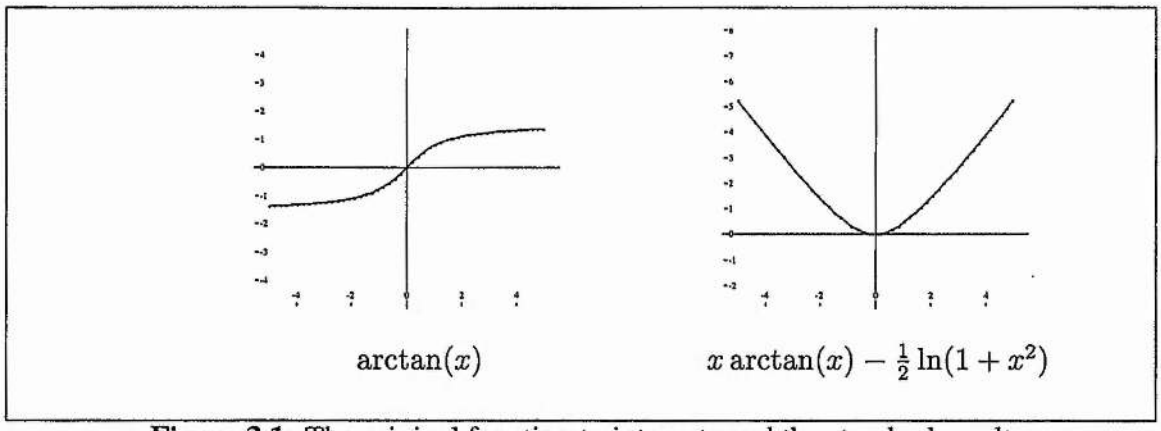


Figure 3.1: The original function to integrate and the standard result

Since there are no discontinuities in arctan the definite integral is easily found using the fundamental theorem of calculus:

$$\forall a, b\,.\,\int_a^b \arctan(x)\,dx = b\arctan(b) - \frac{1}{2}\ln(1+b^2) - a\arctan(a) + \frac{1}{2}\ln(1+a^2)$$

The test results are listed in Table 3.1.

As the integral is symmetric around 0, Examples 2 and 3 have the value 0 which all the systems calculated correctly. In Example 4, the definite integral

$$\int_{-4}^{0} \arctan(x)dx$$

is

$$4\arctan(-4) + \frac{\ln(17)}{2}$$

which is negative. Here we see that axiom gives a different result which is actually positive. axiom's result depends on axiom's use of the equivalence $2\arctan(x) = \arctan\left(\frac{2x}{1-x^2}\right)$, which is only correct for x between -1 and 1 as discussed in Section 2.1.3. Plots of axiom's result and the difference between this and the standard result can be seen in Figure 3.2, which also illustrates that the two are identical on the interval -1 to 1, exactly where axiom's transformation is valid.

Example 5 shows how the difference in the integral of arctan shows up in the integral of a composite function using arctan.

No.	Integral	axiom	Maple	Mathematica	Matlab
1	$\int \arctan(x)dx$	$\frac{-\ln(x^2+1)-x\arctan(\frac{2x}{x^2-1})}{2}$	*sr*	*sr*	*sr*
2	$\int_{-1}^{1} \arctan(x)dx$	*sr*	*sr*	$\frac{\pi-\ln 4-\pi+\ln 4}{4}$ simplifies to 0	*sr*
3	$\int_{-4}^{4} \arctan(x)dx$	*sr*	*sr*	*sr*	*sr*
4	$\int_{-4}^{0} \arctan(x)dx$	$\frac{\ln(17)+4\arctan(\frac{8}{15})}{2} > 0$	*sr*	*sr*	*sr*
5	$\int x\arctan(\frac{1}{x})dx$	$\frac{(x^2+1)\arctan(\frac{2x}{x^2-1})+2x}{4}$	*sr*	*sr*	*sr*

Table 3.1: Definite integration with arctan

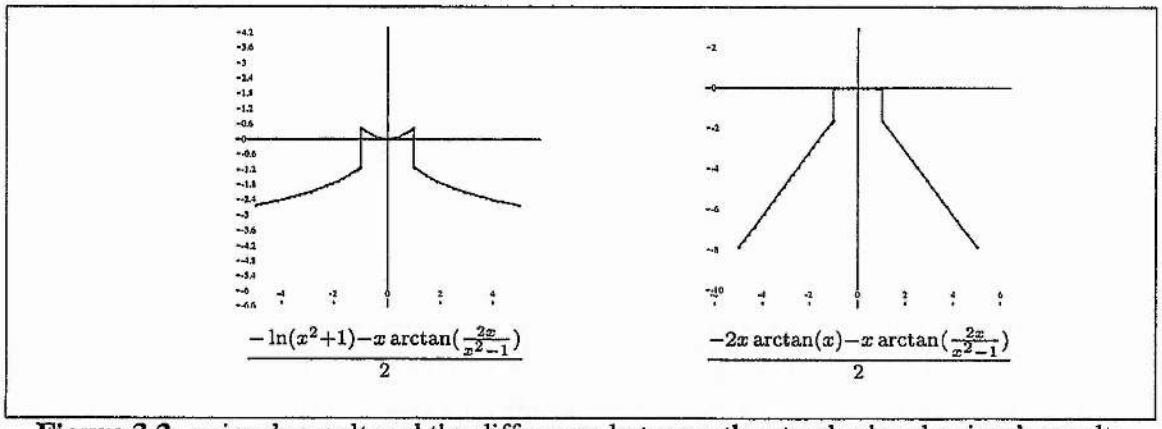


Figure 3.2: axiom's result and the difference between the standard and axiom's results

3.2 Variations over $\frac{1}{x}$

In this section we look at definite integrals of $\frac{1}{x}$. To accommodate negative limits, the indefinite integral $\int \frac{1}{x}dx$ should have value $\ln(|x|)$ rather than just $\ln(x)$ which all the

systems offer as result. So the indefinite integral is

$$\int \frac{1}{x} = \ln|x| + C$$

Since $f(x) = \frac{1}{x}$ has a double pole at 0, any definite integral over 0 is diverging, unless CPV methods apply.

The *standard results* for the definite integral are then:

$$\int_a^b \frac{1}{x}\,dx = \begin{cases} \ln|b| - \ln|a| & \forall a, b < 0 \text{ or } a, b > 0 \\ \text{undefined/diverging} & \text{otherwise} \end{cases}$$

The *naive result* is then

$$\forall a, b\,.\ \int_a^b \frac{1}{x}\,dx = \ln(b) - \ln(a)$$

which is only correct if a and b are both positive.

3.2.1 One-parameter Limits

We first test the four systems using only one parameter in the limits. The test results can be found in Table 3.2.

In Examples 2 and 3, we see that both axiom and Mathematica seem to pick up on the pole and refuse to integrate over it, whereas both Maple and Matlab do not check the parameters in the limits.

The answers returned by axiom, Maple and Mathematica in Example 4 are undefined when $a = 0$, and it appears axiom, Maple and Mathematica may assume $a \neq 0$. In Example 4 we also see several different ways of expressing $\ln(2)$. The fact that axiom expresses its result in terms of a^2 might indicate that it is trying to accommodate for negative values of a, although this is not evident from the indefinite integral.

3.2.2 Two-parameter Limits

We now look at some examples using two parameters in the limits and some related one-parameter examples. As we are using a^2 and b^2 the signs of the parameters are not an issue in these examples.

No.	Integral	axiom	Maple	Mathematica	Matlab
1	$\int \frac{1}{x} dx$	$\ln(x)$	$\ln(x)$	$\ln(x)$	$\ln(x)$
2	$\int_{-a}^{a} \frac{1}{x} dx$	d	$\ln(a) - \ln(-a)$	d	$\ln(a) - \ln(-a)$
3	$\int_{-a}^{2a} \frac{1}{x} dx$	d	$\ln(2a) - \ln(-a)$	d	$\ln(2a) - \ln(-a)$
4	$\int_{a}^{2a} \frac{1}{x} dx$	$\frac{\ln(4a^2)-\ln(a^2)}{2}$	$\ln(2a) - \ln(a)$	$\ln(2a) - \ln(a)$	$\ln(2)$

Table 3.2: Definite integration with $\frac{1}{x}$ and one parameter

The test results can be found in Table 3.3.

From the results of Examples 2 and 5, we see that although Mathematica does some checking of the limits, and knows that a^2 and $-a^2$ are on either side of the pole, it fails to work that out for b^2 and $-a^2$ too. This *could* indicate that Mathematica is using a table for the simpler of the two cases, although this can not be determined from the documentation, nor from the use of Mathematica itself.

3.2.3 Other Limits

In the following the CPV method is used to indicate results obtained when using the Maple option `CauchyPrincipalValue`, this allows equal integrals on either side of a pole to cancel out. It can only be used with limits expressed without parameters. The results can be found in Table 3.4.

From the result of Example 3,

$$\int_{\sqrt{a^2}}^{\sqrt{b^2}} \frac{1}{x} dx$$

No.	Integral	axiom	Maple	Mathematica	Matlab
1	$\int_0^{b^2} \frac{1}{x} dx$	d	d	d	d
2	$\int_{-a^2}^{b^2} \frac{1}{x} dx$	d	n	n	n
3	$\int_{a^2}^{b^2} \frac{1}{x} dx$	sr	sr	sr	sr
4	$\int_{a^2+1}^{b^2} \frac{1}{x} dx$	sr	sr	sr	sr
5	$\int_{-a^2}^{a^2} \frac{1}{x} dx$	d	n	d	n
6	$\int_{-a^2-1}^{a^2} \frac{1}{x} dx$	d	n	n	n

Table 3.3: Definite integration of $\frac{1}{x}$ with one or more parameters

we see that Matlab not only assumes $a,\ b \neq 0$, but that $a,\ b > 0$.

3.3 Variations over $\frac{1}{x-c}$

Next we go on to integrating the function $\frac{1}{x-c}$. The *standard result* of the indefinite integration is then

$$\int \frac{1}{x-c} dx = \ln|x-c| + C$$

and for the definite integral it is

$$\int_a^b \frac{1}{x-c}\, dx = \ln|b-c| - \ln|a-c| \; \forall a, b < c \text{ or } a, b > c$$

No.	Integral	axiom	Maple	Mathematica	Matlab
1	$\int_{-a}^{a+1} \frac{1}{x} dx$	*pp*	*n*	*n*	*n*
2	$\int_{-a}^{b} \frac{1}{x} dx$	*pp*	*n*	*n*	*n*
3	$\int_{\sqrt{a^2}}^{\sqrt{b^2}} \frac{1}{x} dx$	*sr*	*sr*	*sr*	$\ln(b) - \ln(a)$
4	$\int_{-3/4}^{3/4} \frac{1}{x} dx$	*d*	*d* $CPV:$ $-\ln(4) - \ln(-\frac{1}{4}) + i\pi \equiv 0$	*d*	*d*
5	$\int_{-2^2}^{4^2} \frac{1}{x} dx$	*d*	*d* $CPV:\ \ln(16) - \ln(4)$	*d*	*d*

Table 3.4: More integration with $\frac{1}{x}$

The *naive result* of the definite integral is

$$\int_a^b \frac{1}{x-c} dx = \ln(b-c) - \ln(a-c)$$

Depending on the values of a and b this might be correct.

For Examples 1–5 we specialise each of these results with $c = 1$, so that the *standard result* of the indefinite integration is

$$\int \frac{1}{x-1} dx = \ln|x-1| + C$$

and of the definite integral it is

$$\int_a^b \frac{1}{x-1}\, dx = \ln|b-1| - \ln|a-1| \,\forall a, b < 1 \text{ or } a, b > 1$$

The *naive result* of the definite integral is then

$$\int_a^b \frac{1}{x-1}dx = \ln(b-1) - \ln(a-1)$$

The test results can be found in Table 3.5.

No.	Integral	axiom	Maple	Mathematica	Matlab
1	$\int \frac{1}{x-1}dx$	*sr*	*sr*	*sr*	*sr*
2	$\int_{-a}^{a} \frac{1}{x-1}dx$	*pp*	*n*	*n*	*n*
3	$\int_a^b \frac{1}{x-1}dx$	*pp*	*n*	*n*	*n*
4	$\int_1^b \frac{1}{x-1}dx$	*d*	*d*	*d*	*d*
5	$\int_0^b \frac{1}{x-1}dx$	*pp*	$\ln(b-1) - i\pi$	$\ln(b-1) - i\pi$	$\ln(b-1) - i\pi$
6	$\int_{a-b}^{a+b} \frac{1}{x-a}dx$	*d*	*n*	*d*	*n*
7	$\int_{a-b}^{a+2b} \frac{1}{x-a}dx$	*d*	*n*	*d*	*n*
8	$\int_{a-b}^{a+c} \frac{1}{x-a}dx$	*pp*	*n*	*n*	*n*

Table 3.5: Definite integration with $\frac{1}{x-1}$

In Example 5 we suddenly see $i\pi$ appearing. This is as a result of evaluating the naive

result, thus getting $\ln(-1)$ which is exactly $i\pi$.

3.4 Variations over $\frac{1}{x^2+a}$

We now present results of some variations of $\frac{1}{x^2+a}$. The *standard result* of integrating this is:

$$\int \frac{1}{x^2+a}dx = \begin{cases} \frac{\arctan\left(\frac{x\sqrt{a}}{a}\right)}{\sqrt{a}} + C & \text{if } a > 0 \\ -\frac{1}{x} + C & \text{if } a = 0 \\ \frac{\ln\left|\frac{x-\sqrt{-a}}{x+\sqrt{-a}}\right|}{2\sqrt{-a}} + C & \text{if } a < 0 \end{cases}$$

axiom gives the two solutions corresponding to non-zero values of a, and we will note that as the *standard result* in this case.

For the definite integrals the *standard results* are:

$$\int_b^c \frac{1}{x^2+a}\,dx = \begin{cases} \frac{\arctan\left(\frac{c\sqrt{a}}{a}\right)}{\sqrt{a}} - \frac{\arctan\left(\frac{b\sqrt{a}}{a}\right)}{\sqrt{a}} & \text{if } a > 0 \\ \frac{1}{b} - \frac{1}{c} & \text{if } a = 0 \wedge (b, c > 0 \vee b, c < 0) \\ \frac{\ln\left|\frac{c-\sqrt{-a}}{c+\sqrt{-a}}\right|}{2\sqrt{-a}} - \frac{\ln\left|\frac{b-\sqrt{-a}}{b+\sqrt{-a}}\right|}{2\sqrt{-a}} & \text{if } a < 0 \\ & \wedge(b, c > \sqrt{-a} \vee b, c < \sqrt{-a}) \\ \text{diverging} & \text{otherwise} \end{cases}$$

Again, axiom gives the first and third cases corresponding to some of the non-zero values of a, and we will note that as the *standard result* in this case.

The *naive result* is taken to be the one simply assuming $a > 0$ at all times and ignoring any possible poles, i.e. the first case. The test results can be found in Table 3.6.

So where there are more possible results axiom will list them. However it does not tell us which result corresponds to which value of a. In some cases this might be difficult to detect. Again Maple, Mathematica and Matlab simply assume that $a > 0$.

No.	Integral	axiom	Maple	Mathematica	Matlab
1	$\int \frac{1}{x^2} dx$	*sr*	*sr*	*sr*	*sr*
2	$\int_{-1}^{1} \frac{1}{x^2} dx$	*d*	*d*	*d*	*d*
3	$\int_{-1/2}^{\pi} \frac{1}{x^2} dx$	*pp*	*d*	*d*	*d*
4	$\int \frac{1}{x^2+a} dx$	*sr*	*n*	*n*	*n*
5	$\int \frac{1}{x^2-2} dx$	*sr*	*n*	*n*	$\frac{1}{2}\sqrt{2}\text{atanh}(\frac{1}{2}x\sqrt{2})$
6	$\int \frac{1}{x^2-a} dx$	*sr*	*n*	*n*	*n*
7	$\int_{b}^{c} \frac{1}{x^2+a} dx$	*pp*	*n*	*n*	*n*

Table 3.6: Definite integration with $\frac{1}{x^2+a}$

3.5 Summary

In this chapter we have demonstrated how even on very simple examples CASs give a variety of answers that are variously completely incorrect, incorrect without additional assumptions, or even if correct are not obviously so as they are equivalent, but not identical to, the standard results.

In general axiom does better at finding poles and displaying the appropriate error messages. In most circumstances Mathematica will give an error whenever there is definitely a pole

within the range, but when there exist parameters so that the integral is defined, Mathematica generally gives a naive answer. When using parameters in the limits of integration, Maple can not pick up any problems, whereas when using actual values Maple refuses to do the integration over a pole unless the option `CauchyPrincipalValue` is used in order to have parts of the integral around the pole cancelled out. Matlab also only picks up poles when using values as limits.

All the examples we have studied in this chapter are very simple, and even so the CASs give confusing and ambiguous, if not plain wrong, results. Larger applications are only likely to have an even greater risk of incorrect or only partially correct results.

Chapter 4

PVS

In this chapter we describe in some detail various aspects of PVS [OSRSC99b], which is a specification and verification tool based on classical, higher order logic. It consists of a specification language, various predefined theories and a theorem prover which supports a high level of automation. The specification language is strongly typed and supports predicate and dependent sub-typing.

Our emphasis is on support for real analysis, so we will look specifically at how the real numbers are implemented in PVS and what operations on them are supported (Section 4.2). Also, Dutertre developed a real analysis library [Dut96] of which we give an overview in Section 4.3. Dutertre's implementation gives a foundation for analysis with respect to convergence and limits, continuity and differentiation of real-valued functions, however the definition of convergence is not equivalent to the usual definition, and so leads to undesirable results. Through working extensively with Dutertre's library we were able to identify and correct this error. In Section 4.4 we describe in some detail how we can use PVS's strong type system, particularly how type judgements can be used to improve matching within existing strategies. Section 4.5 explains how one can define new strategies specific to an application.

4.1 Introduction

PVS is used extensively in both hardware and software verification. Two examples are verification of the Airborne Information for Lateral Spacing (AILS) Alerting Algorithm (Section 2.3.3), which is concerned with the correctness of warning algorithms used when aircrafts land on parallel runways [CM00], and development of a fault-tolerant clock synchronisation circuit with optimisations verified in PVS [Min98].

The core of PVS is implemented in Allegro Common Lisp. It is usually run via an extensive emacs interface which allows for short-cuts for commands and – via Tcl/Tk – supports graphical representations of proof trees and library structures. However, PVS can also be run in batch mode or, as discussed in Section 2.4, from other systems via a basic Tcl/Tk interface [ADG+01].

4.1.1 Specifications

In PVS, specifications are organised into *theories*. The structure of a typical theory is shown in Figure 4.1. A theory may be parameterised, so that the `example_theory` takes a parameter `T` which is a subtype of `real` (the real numbers in PVS). This allows abstraction from types and/or parameters used in the theory, for example a theory implementing intervals over $\mathbf{R}$ might have the parameters

```
[ a:  real, b:  { x:  real | a <= x } ]
```

and then define and work with the closed interval $[a, b]$.

Assumptions may be made, for instance about `T` in our `example_theory` where we assume that `T` is connected. When we instantiate the theory to use it, we will need to make sure that the actual parameter is itself connected, PVS ensures this by generating a Type-Correctness Condition (TCC, Section 4.4) for us to prove. During proofs in the `example_theory` we can use the assumption as an ordinary theorem.

Within theories we can now declare variables, such as

```
f: VAR [ T -> real ]
```

```
example_theory [ T: TYPE FROM real ]: THEORY

  BEGIN

  ASSUMING
    connected: ASSUMPTION
      FORALL (x,y: T): FORALL (z: real): x < z AND z < y
        IMPLIES T_pred(z)
  ENDASSUMING

  IMPORTING limit_of_functions[T], real_fun_ops_ext

  f, f1, f2: VAR [ T -> real ]
  u: VAR real
  x, y: VAR T

  g(f, x, y): real = f(x) + f(y)

  theorem1: THEOREM
    < boolean expression >

  lemma1: LEMMA
    < boolean expression >

  END example_theory
```

Figure 4.1: Structure of PVS theory

that is, `f` is a function from `T` to `real`. As `f` is just declared, not defined, we say that it is *uninterpreted.*

We can also define functions and predicates, for example using recursion as in

```
fac(n): RECURSIVE nat =
  IF n = 0 THEN 1
  ELSE n * fac(n-1)
```

```
  ENDIF
MEASURE n
```

The MEASURE is required with recursive definitions and is used by PVS to ensure the recursion is well founded [OSRSC99a].

Finally we have the formulae of the theory. They can be stated using various keywords such as `proposition`, `formula`, `lemma` and `theorem` [OSRSC99a]. They all have the same semantics and the different keywords are provided to allow the user to distinguish between different types of formulae. In our work we mainly use `lemma` and `theorem`, and in the context of PVS we will use the two terms as being totally synonymous.

Within theories we can also declare type judgements, which provide a means of stating that expressions are of a certain type. This facility will be explained further in Section 4.4.

4.1.2 Predefined Theories

The built-in theories for PVS (such as base types like the booleans and the real numbers and composite types like sequences) can be found in the file `prelude.pvs` (provided as part of the PVS distribution and loaded at initialisation of a PVS session). These theories are always loaded when running PVS and contain essential declarations and definitions, for instance of types.

One of the operators provided in the prelude is the choice operator `epsilon`. When given a predicate over a type `T`, `epsilon` returns an element of `T` satisfying the predicate if one exists, otherwise `epsilon` returns an arbitrary element of `T`. This means that `epsilon` always returns an answer. The type `T` is required to be non-empty so when using the choice operator a TCC will be generated requesting a proof that `T` is non-empty.

The prelude not only provides many types and their related operations, but also serves as a repository of examples of the various facilities in the specification language.

4.1.3 Theorem Prover

The prover facilities [SORSC99] of PVS support a high level of automation in the form of a large collection of proof rules and strategies, and a strategy language for users to define their own strategies (Section 4.5). In general the automation works quite well, provided one knows which arguments to give the various strategies. The most well-known PVS strategy is `grind`. It works as a brute-force search for proofs, performing repeated Skolemisation, (BDD-)simplification, rewriting, and application of theorems. By choosing arguments to `grind` carefully one can ensure a fairly high rate of success, however using `grind` – or any other high level strategy – will often fail due to the problem of instantiating the universally quantified variable(s) of applicable theorems correctly. This is an inherently difficult question which theorem provers in general struggle with.

During the application of higher level strategies PVS might try to match the current goal to an existing theorem. An application of a theorem can generate TCCs and PVS will try to discharge these. However, `grind` for example will not automatically apply the theorem unless the TCCs can be discharged automatically. This differs from when we use a lower level of automation and apply the theorem directly, then any TCCs generated from the matching are passed back for the user to prove.

4.2 Real Numbers in PVS

The real numbers is one of the base types in PVS and so is included in the prelude. It is denoted by `real`. The implementation is based on an axiomatisation which is embedded in the typechecker and the theorem prover. So for example, PVS "knows" that

$$0,\ 1,\ 2,\ 3,\ \ldots \in \mathbf{R}$$
$$0 \neq 1,\ 0 \neq 2,\ 1 \neq 2,\ \ldots$$

With the real numbers in place there is an explicit declaration of the subtype nonzero-real

```
nonzero_real: NONEMPTY_TYPE = {r: real | r /= 0} CONTAINING 1
```

and its abbreviation `nzreal`.

Then there are definitions of the usual operators in the reals:

```
+, -, *: [ real, real -> real ]
/:       [ real, nzreal -> real ]
-:       [ real -> real ]

<(x, y):  bool
<=(x, y): bool = x < y OR x = y;
>(x, y):  bool = y < x;
>=(x, y): bool = y <= x
```

where the order-operators may be used infix as well. Again, knowledge about the order of 0, 1, 2, ... is built into the typechecker and theorem prover.

Next the prelude includes axioms stating rules such as commutativity, associativity and distributivity. These axioms are introduced using the keyword POSTULATE to indicate that they are provable by the decision procedures but not from other axioms.

```
commutative_add:  POSTULATE x + y = y + x
associative_add:  POSTULATE x + (y + z) = (x + y) + z

commutative_mult: POSTULATE x * y = y * x
associative_mult: POSTULATE x * (y * z) =  (x * y) * z

distributive:     POSTULATE x * (y + z) = (x * y) + (x * z)
```

Finally, based on explicit definitions of (least) upper bound and (greatest) lower bound, an axiom states completeness of the reals in the sense that any non-empty set of reals which is bounded above has a least upper bound:

```
real_complete: AXIOM
  FORALL S:
    (EXISTS y: upper_bound?(y, S)) IMPLIES
      (EXISTS y: least_upper_bound?(y, S))
```

```
both_sides_plus1:  LEMMA (x + z = y + z) IFF x = y
both_sides_times1: LEMMA (x * n0z = y * n0z) IFF x = y
times_div1:        LEMMA x * (y/n0z) = (x * y)/n0z
div_times:         LEMMA (x/n0x) * (y/n0y) = (x*y)/(n0x*n0y)
div_eq_zero:       LEMMA x/n0z = 0 IFF x = 0
div_simp:          LEMMA n0x/n0x = 1
cross_mult:        LEMMA (x/n0x = y/n0y) IFF (x * n0y = y * n0x)
zero_times3:       LEMMA x * y = 0 IFF x = 0 OR y = 0
```

Figure 4.2: Cancellation laws for equality

Based on these axioms and the implementation of the typechecker and theorem prover, the rest of the development of `real` is proven rather than axiomatised.

First, there is a collection of judgements (see Section 4.4) about the various operators and how they preserve or change the types of their arguments, for example

```
px, py: VAR posreal

posreal_times_posreal_is_posreal: JUDGEMENT *(px, py) HAS_TYPE posreal
```

which states that the product of two positive reals, px and py, is a positive real.

Also proven is a large collection of cancellation laws for equality, some of which are given in Figure 4.2, and order lemmas and cancellation laws for $<$, some of which can be seen in Figure 4.3. Similar lemmas exists for each of $\leq, >$ and $\geq$.

PVS also has an exponentiation function. It is restricted to take non-negative integer powers of real numbers. With this definition there is a limited selection of theorems about the power function such as $x^n > 0$ for $x > 0$ and $x^n \neq 0$ for $x \neq 0$.

So all the usual operations and cancellation laws for the real numbers are available in PVS, providing a good foundation for doing arithmetic with the real numbers. However, it is still lacking similarly simple lemmas which are useful for doing analysis, for example $|c| < 1 \Rightarrow |c| * x < x$ for $x > 0$, but lemmas such as these can be proven quite easily based on what is already available. Much of the basic axiomatisation is built into the typechecker and theorem prover, this in particular means that one does not usually need to invoke any

```
trich_lt:          LEMMA x < y OR x = y OR y < x
neg_lt:            LEMMA 0 < -x IFF x < 0
pos_times_lt:      LEMMA 0 < x * y
  IFF (0 < x AND 0 < y) OR (x < 0 AND y < 0)
neg_times_lt:      LEMMA x * y < 0
  IFF (0 < x AND y < 0) OR (x < 0 AND 0 < y)
pos_div_lt:        LEMMA 0 < x/n0y
  IFF (0 < x AND 0 < n0y) OR (x < 0 AND n0y < 0)
neg_div_lt:        LEMMA x/n0y < 0
  IFF (0 < x AND n0y < 0) OR (x < 0 AND 0 < n0y)
div_mult_pos_lt1: LEMMA z/py < x IFF z < x * py

both_sides_plus_lt1: LEMMA x + z < y + z IFF x < y
lt_plus_lt1:         LEMMA x <= y AND z < w IMPLIES x + z < y + w
lt_minus_lt1:        LEMMA x <= y AND w < z IMPLIES x - z < y - w
lt_div_lt_pos1:      LEMMA px <= y AND pz < w IMPLIES px/w < y/pz
lt_div_lt_neg1:      LEMMA x <= ny AND z < nw IMPLIES ny/z < x/nw
```

Figure 4.3: Order and cancellation laws for $<$

of the axioms during proofs, however it also means that we do not actually know how the reals are implemented.

4.3 Existing Analysis Library

PVS's definition of the reals forms the basis of Dutertre's library [Dut96] for basic real analysis. This library contains many basic definitions and theorems used in real analysis, including sequences of reals, convergence of functions and of sequences, and continuity and differentiation of real-valued functions. Below we briefly outline the contents of the library for each of these areas of real analysis.

4.3.1 Sequences of Reals

The PVS prelude implements a polymorphic sequence, which Dutertre specialised to sequences of real numbers, proving theorems about properties of sequences of reals such as increasing or decreasing, extracting a subsequence and how subsequences inherit properties such as boundedness. This part of the library also contains a theory defining convergence of sequences

```
u : VAR sequence[real]
l : VAR real
epsilon : VAR posreal
i, n : VAR nat

convergence(u, l) : bool =
  FORALL epsilon : EXISTS n :
    FORALL i : i >= n IMPLIES abs(u(i) - l) < epsilon
```

It also gives various criteria for a sequence to be convergent, such as the Squeeze Theorem. Finally the limits of the usual combinations of sequences are given, for example the limit of $s_1(n) + s_2(n)$ is the sum of the limits of $s_1(n)$ and $s_2(n)$.

4.3.2 Limits of Functions

Dutertre used ε–δ to define convergence and thereby limits of functions, however the definition (shown in Figure 4.4) is somewhat unusual, leading to results which do not fit with our usual understanding of analysis. The predicate `convergence` is applied to a function f, a set E, which may or may not be a subset of the domain of f, a real number a, which is the point we wish to check for a limit and another real l which is the proposed limit. There are essentially two problems with this definition. The first is that the conditions on x allows $x = a$, so that the requirements are too strong, and closely resembles those of continuity. This is also proven by Dutertre's "theorem"

```
continuity_def2: THEOREM
  continuous(f, x0) IFF convergent(f, x0)
```

```
epsilon, delta, e: VAR posreal
E: VAR setof[real]
f: VAR [ T -> real ]
a, l, z: VAR real
x: VAR T

adh(E): setof[real] =
  { z | FORALL e: EXISTS x: E(x) AND abs(x - z) < e }

convergence(f, E, a, l): bool =
  adh(E)(a) AND FORALL epsilon: EXISTS delta: FORALL x:
    E(x) AND abs(x - a) < delta IMPLIES abs(f(x) - l) < epsilon
```

Figure 4.4: Dutertre's definition of convergence of functions

Furthermore, Dutertre only requires a to be adherent to E, whereas the usual definitions also require a to be an accumulation point in E, that is "a is not isolated in E" [Apo74, Mad91].

Finding this inconsistency with the usual notion of convergence was not easy, and it is a good example of how theorem provers are no more correct than the specification the user starts out with. Of course, a developer using theorem proving is free to give definitions and theorems any names, and so in a sense it is not appropriate to deem Dutertre's definition of convergence *wrong*, as the definition is perfectly acceptable. However, it does not correspond to *the usual notion* of convergence. If the user community has an agreed meaning of a certain word, this should be respected in the formalisation. Thus without any further explanation, the naming of properties and theorems should reflect their use outside theorem provers.

Both errors are corrected in our version of the definition of convergence (Figure 4.5).

As for sequences of real numbers, Dutertre proved theorems about preserving convergence under addition, subtraction, multiplication, division and function composition, that is convergence of rational functions. Having corrected the definition of convergence we proved those theorems again. In general, the proofs did not have to be changed much, as the added restrictions are on the domain of the function and the point of interest. Finally, various

```
epsilon, delta, e: VAR posreal
E: VAR setof[real]
f: VAR [ T -> real ]
a, l, z: VAR real
x, y: VAR T

adh(E): setof[real] =
  { z | FORALL e: EXISTS (x: real): E(x) AND abs(x - z) < e }

not_isolated_point(a,E): bool =
  FORALL (e: posreal): EXISTS (y: real):
    E(y) AND ((a < y AND y < a + e) OR (a - e < y AND y < a))

convergence(f, E, a, l): bool =
  adh(E)(a) AND not_isolated_point(a, E) AND
    FORALL epsilon: EXISTS delta:
      FORALL x: E(x) AND x /= a AND abs(x - a) < delta
        IMPLIES abs(f(x) - l) < epsilon
```

Figure 4.5: Corrected definition of convergence of functions

bounds on limits are given.

4.3.3 Continuous Functions

When defining continuity of functions, Dutertre refers back to the definition of convergence, but then proves that the definition is equivalent to the standard ε–δ definition of continuity. As for limits, the library contains lemmas about operations which preserve continuity, see Figure 4.6, so again Dutertre is working with rational functions. There is also a theorem about function composition preserving continuity, that is if $f : T_1 \to T_2$ is continuous at x_0 and $g : T_2 \to \mathbf{R}$ is continuous at $f(x_0)$, then $g(f(x))$ is continuous at x_0 too.

By considering a continuous function restricted to a subinterval of its domain further theorems are proved. For example, Dutertre proves the intermediate value theorem for contin-

```
f1, f2: VAR [ T -> real ]
g:      VAR [ T -> nzreal ]
h:      VAR [ T1 -> T ]
x0:     VAR T
x:      VAR T1

sum_continuous: THEOREM continuous(f1, x0) AND continuous(f2, x0)
  => continuous(f1 + f2, x0)

diff_continuous: THEOREM continuous(f1, x0) AND continuous(f2, x0)
  => continuous(f1 - f2, x0)

prod_continuous: THEOREM continuous(f1, x0) AND continuous(f2, x0)
  => continuous(f1 * f2, x0)

const_continuous: THEOREM continuous(k, x0)

scal_continuous: THEOREM continuous(f1, x0) => continuous(k * f1, x0)

opp_continuous: THEOREM continuous(f1, x0) => continuous(- f1, x0)

div_continuous: THEOREM continuous(f1, x0) AND continuous(g, x0)
  => continuous(f1 / g, x0)

inv_continuous: THEOREM continuous(g, x0) => continuous(1 / g, x0)

identity_continuous: THEOREM continuous(I[T], x0)

abs_continuous: THEOREM
  continuous(f1, x) => continuous(abs(f1), x)

comp_continuous: THEOREM continuous(h, x) AND continuous(f2, h(x))
  => continuous(f2 o h, x)
```

Figure 4.6: Dutertre's lemmas about preservation of continuity

uous functions [Apo74]:

Theorem 4.1 (Intermediate Value Theorem for Continuous Functions) *Let $a < b$ and $f : [a, b] \to \mathbf{R}$ with f continuous. Suppose $f(a) \neq f(b)$, then f takes every value between $f(a)$ and $f(b)$ on the interval (a, b).*

In PVS, this can then be stated using the following two lemmas (it is assumed that T is connected):

```
f: VAR [ T -> real ]
g: VAR { f | continuous(f) }
x, y, z: VAR real

intermediate1: PROPOSITION
  a <= b AND g(a) <= x AND x <= g(b) IMPLIES
    EXISTS c: a <= c AND c <= b AND g(c) = x

intermediate2: PROPOSITION
  a <= b AND g(b) <= x AND x <= g(a) IMPLIES
    EXISTS c: a <= c AND c <= b AND g(c) = x
```

4.3.4 Differentiation

Again Dutertre uses the standard definition using the Newton quotient:

> A function f is differentiable if, and only if,
>
> $$\frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{4.1}$$
>
> has a limit as Δx tends to 0.

The fact that a differentiable function is continuous is established and just as for continuity we have all the usual rules for combining functions and preserving differentiability, including the values of the derivatives. As for continuity, Dutertre is working with rational functions. The value of the derivative at a maximum or a minimum is also given, as is the mean value theorem:

```
mean_value: THEOREM derivable(f) AND a < b
  => EXISTS c: a < c AND c < b AND deriv(f, c) * (b - a) = f(b) - f(a)
```

4.3.5 Roots

In addition to the analysis library Dutertre has also built a theory to handle roots. This implementation covers positive integer roots of nonnegative reals and provides a useful extension of the power functions native to PVS as it provides a way to handle rational powers. So one would express $x^{\frac{3}{2}}$ as `expt(root(x, 2), 3)` or `root(expt(x, 3), 2)`. These two are proved equivalent by Dutertre.

Just as the prelude contains lemmas about order and cancellation when using simple arithmetic and exponentiation, Dutertre added lemmas about order and cancellation with roots. Some of these are:

$$\forall m \in \mathbf{N}_+ \,.\, \sqrt[m]{0} = 0$$
$$\forall m \in \mathbf{N}_+ \,.\, \sqrt[m]{1} = 1$$
$$\forall x \in \mathbf{R} \setminus \mathbf{R}_- \,.\, \sqrt[1]{x} = x$$
$$\forall x \in \mathbf{R} \setminus \mathbf{R}_-,\, m \in \mathbf{N}_+ \,.\, (\sqrt[m]{x})^m = x$$
$$\forall x \in \mathbf{R} \setminus \mathbf{R}_-,\, m \in \mathbf{N}_+ \,.\, \sqrt[m]{x^m} = x$$
$$\forall x \in \mathbf{R} \setminus \mathbf{R}_-,\, m \in \mathbf{N}_+ \,.\, \sqrt[m]{x} = 0 \Leftrightarrow x = 0$$
$$\forall x,\, y \in \mathbf{R} \setminus \mathbf{R}_-,\, m \in \mathbf{N}_+ \,.\, \sqrt[m]{x * y} = \sqrt[m]{x} * \sqrt[m]{y}$$
$$\forall x,\, y \in \mathbf{R} \setminus \mathbf{R}_-,\, m \in \mathbf{N}_+ \,.\, \sqrt[m]{x} = \sqrt[m]{y} \Leftrightarrow x = y$$
$$\forall x,\, y \in \mathbf{R} \setminus \mathbf{R}_-,\, m \in \mathbf{N}_+ \,.\, \sqrt[m]{x} < \sqrt[m]{y} \Leftrightarrow x < y$$
$$\forall x \in \mathbf{R} \setminus \mathbf{R}_-,\, m \in \mathbf{N}_+ \,.\, x < 1 \Rightarrow \sqrt[m]{x} < 1$$

4.4 Types and Judgements in PVS

The specification language for PVS is strongly typed. Of particular interest for our work are the base types `number` (with subtypes `real`, `rat`, `int` and `nat`) and `boolean` and function types. For example the predicate `convergence` about limits of functions from Section 4.3 has the type `[ (T -> real, setof[real], real, real) -> bool ]`. By using pa-

rameterised theories as described in Section 4.1 we can construct abstract datatypes, for example the prelude file contains a theory

```
sequences[ T: TYPE ]: THEORY
```

which implements an abstract datatype of sequences. This can then be instantiated with for instance `real` to give a theory of sequences of reals, just as Dutertre did in `convergence_sequences` which implements convergence of real-valued sequences.

PVS supports two different ways of declaring subtypes. Either by using a boolean expression as in

```
negreal: TYPE = { x: real | x < 0 }
```

which declares `negreal` to be the type of negative reals, this is called predicate sub-typing, or by declaring an uninterpreted subtype as in

```
s: TYPE from real
```

which declares `s` to be some subtype of `real`.

Since the user can give arbitrary boolean expressions in type declarations typechecking is undecidable. Therefore Type-Correctness Conditions (TCCs) are generated during typechecking. Some of these might be discharged by PVS automatically, but others might be left for the user to prove. For example, typechecking the function definition

```
div1(x: negreal): real = 1 / x
```

raises this TCC

```
div1_TCC1: OBLIGATION (FORALL (x: negreal): x /= 0)
```

This is because division is of the type `[ real, nzreal -> real ]`. The TCC is discharged automatically by PVS, as the type `negreal` is known to be a subtype of `nzreal` as we shall now see.

Type judgements are used to help the typechecker discharge some of the many TCCs that might occur when using sub-typing. Considering again the function `div1` from above, we need a judgement asserting that `negreal` is not only a subtype of `real` but indeed a subtype of `nzreal`. The following judgement (from the `real_types` theory in the PVS prelude) does just that:

```
negreal_is_nzreal: JUDGEMENT negreal SUBTYPE_OF nzreal
```

The judgement is then used by the typechecker, and so the user will not be asked to prove the TCC `div1_TCC1`.

So type judgements are used by PVS when typechecking for example a theory. However, they are also used during proofs. Say we would like to prove the following:

```
f:    VAR [ real -> real ]
g:    VAR [ real -> posreal ]
E:    VAR setof[real]
x, l: VAR real

example: THEOREM
  convergence(f / g, E, x, l)
```

Typechecking of `example` then generates the TCC:

```
example_TCC1: THEOREM
  FORALL (x: real): g(x) /= 0
```

It is automatically discharged by PVS, because g is known to be non-zero valued. But say we wish to use the theorem

```
convergence_div: PROPOSITION
  FORALL E, f, g, a, l1, l2:
    convergence(f, E, a, l1)
      AND convergence(g, E, a, l2)
      AND l2 /= 0
    IMPLIES convergence(f/g, E, a, l1/l2)
```

to prove our theorem `example`. In order to apply `convergence_div` we must ensure that the type fits. This is clearly true, and application of `convergence_div` generates a TCC like that above.

Judgements are written into theories, and when the theory is typechecked a TCC corresponding to the judgement is generated. This can then be proven using the full power of PVS. So a judgement is as a theorem in that it can be applied as a step in a proof and it is formally proven to be correct, however judgements are also used by the typechecker and during matching in the theorem prover. For PVS to apply theorems automatically provable type correctness is essential, and thus type judgements are a powerful tool. In Section 6.4 we will give examples of how judgements are used in this way to check functions for continuity.

4.5 Strategies

The PVS prover contains high level proof commands but also supports a *strategy language* which allows users to write their own proof strategies [SORSC99]. The strategy language is based on Lisp, and the general structure of a strategy is

```
(defstep <name> (<arguments>)
   <expression>
   <documentation>
   <format>
)
```

where `<name>` is the name of the strategy, `<arguments>` can contain required and/or optional arguments or be empty, `<expression>` programs the action of the strategy and `<documentation>` gives any documentation information, which is then made available via the PVS help system. Finally `<format>` is printed whenever the strategy is used, which helps the user to keep track of the progress of the proof. For low-level strategies this is often left empty to avoid too many details being printed.

Example 4.1 *In this example we look at a theory* `bar` *and a strategy* `foo` *to be used with that theory:*

```
bar  [ T: TYPE ]: THEORY
  BEGIN
  f: VAR [ T -> T ]
  bar-lemma: THEOREM
       *some theorem*
  END bar
```

This theory has the parameter T, *which is then used in the declaration of the function* f. *We now would like a strategy which, given a type with which to instantiate* bar, *tries to apply* bar-lemma *to the current goal.*

```
(defstep foo (foo-arg)
  (let (foo-lemma (format nil "bar-lemma[~a]" foo-arg))
       (TRY (lemma foo-lemma) (GRIND) (SKIP)))
 ''If bar-lemma[foo-arg] succeeds and produces sub-goal(s)
   then run grind otherwise skip''
 ''Tries to apply bar-lemma with the right theory instantiation''
)
```

The name of this strategy is foo, *it takes one argument* foo-arg. *If we wish to apply the strategy when using functions of for instance type* [bool -> bool] *we must specify that we would like the actual parameter of* bar *to be* bool. *This is handled by* foo-arg, *when the strategy is used as follows:*

```
(foo ''bool'')
```

The first part of the let-expression names bar-lemma[bool] foo-lemma. *The second part is an application of the built-in strategy* TRY, *which tries to apply the first argument to the current sub-goal. If* TRY *is successful it then applies the second argument to each new sub-goal, if not – it applies the third argument instead. Thus* foo *is a tactic, whereas* TRY *is a tactical.* □

Within a strategy it is possible to inspect the current goal. One can then decide what to do next based on the goal rather than guessing (using TRY or similar) as seen in Example 4.1.

This provides the strategy writer with means to check the type and values of components of the goal before applying any theorems or strategies, thus ensuring that the strategy will not crash due to mismatched types or similar incompatibilities.

PVS also allows *glassbox versions* of strategies. They are invoked by appending $ to the strategy name, as in `foo$`. Using the glassbox version of a strategy causes large amounts of documentation to be printed during the proof, and so is very useful for debugging.

By writing application specific strategies it is possible to gain a very high level of control over proofs and keep them automatic at the same time. However, in many cases the high level proof commands available in PVS are suitable on their own.

4.6 Summary

PVS is a specification and theorem proving tool, which is used for verifying both hardware and software. In the `prelude` file PVS has theories which support the base types such as real numbers, booleans and composite polymorphic types such as sequences. As an extension to the built-in support for reals and functions, Dutertre has implemented a basic real analysis library. This includes definitions of and theorems about convergence of functions and sequences, continuity and differentiability, and also provides support for the use of roots. In general this library follows standard developments as found in mathematics textbooks, however the definition of convergence of functions was incorrect. We identified and corrected this problem, and reproved all theorems affected by this.

Since PVS is strongly typed, type judgements turns out to be very powerful in assisting PVS in both typechecking theories and input to the theorem prover and in matching used internally in the theorem prover. Also, PVS supports user defined strategies with a Lisp-based strategy language. This allows users to write special-purpose strategies.

Chapter 5

Transcendental Functions in PVS

In this chapter we describe our development of transcendental functions such as exp, cos and sin in PVS. The main motivation for this implementation is that we wish to use PVS to support CASs, for example in discharging side conditions (see Section 2.1.2) and in doing automatic continuity checking (see Chapter 6). In order to do this in the presence of transcendental functions we need not only to implement definitions of some transcendental functions but also a large database of lemmas about those functions. In Section 5.1 we give an overview of Harrison's analysis library in HOL Light [Har98] and an overview of our own development in PVS. Section 5.2 explains our implementation of infinite series and Section 5.3 gives an overview of our implementation of some transcendental functions. In Section 5.4 we describe the implementation of continuity and differentiation, in particular with respect to power series. Finally, in Section 5.5 we compare our implementation to the axiomatisation of trigonometric functions done at ICASE/NASA Langley [MCB+].

As our library is based on Dutertre's real analysis library (see Section 4.3), we have included in Appendix C a listing of all the files in our library, clearly showing which files are Dutertre's original files (and whether or not we have modified them) and which files are new. All the PVS theories, proofs and strategies are supplied on the CD-ROM.

Most of our definitions, and in particular the lemmas, are equivalent to Harrison's. Our development rests on Dutertre's analysis library which uses ε–δ definitions as the basis for convergence and continuity, where Harrison's library uses the more general notion of convergence nets, thus the proofs in the two implementations are quite different. However, for the end user of the libraries the implementations look very similar, modulo the different

top-level notation of PVS and HOL Light.

5.1 Introduction

We wish to implement the transcendental functions exp, cos and sin and their inverses in PVS. We have already seen how Dutertre implemented support for doing real analysis (such as continuity and differentiability) for rational functions with parameters, that is functions made up of the identity functions, constants and the combinators $+$, $-$ (unary as well as binary), $*$ and $/$. We wish to extend this to elementary functions, that is combinations of transcendental and rational functions, such as

$$f(x) = \frac{\cos(x)}{\exp(x + a)}$$

In high school the trigonometric functions are defined geometrically using triangles and angles, but one can also define them by certain power series. This allows for analytical treatment of them. Following standard analysis, we start with a theory of partial sums and then consider sequences of these to get infinite series. Power series are then obtained by specialising infinite series. We define transcendental functions by their power series and via the power series prove a collection of theorems about the functions.

The implementation described in this chapter is based on the analysis library developed by Dutertre [Dut96] with our revisions, as described in Section 4.3.

5.1.1 HOL Development

Harrison implemented an advanced real analysis library in HOL Light [Har98]. Parts of this library served as inspiration for our PVS implementation, so we will now highlight some of the differences between Harrison's HOL Light implementation and Dutertre's in PVS.

Real Numbers Harrison's implementation of the reals is a variation of Cantor's method which identifies a real number with a set of rational sequences that converge to it, however Harrison scales the terms of the sequences to natural numbers.

A *Cauchy sequence* is a sequence x_n with the following property:

$$\forall \varepsilon > 0\, \exists N\, \forall n,\, m \geq N\, .\, |x_m - x_n| < \varepsilon$$

Harrison then considers Cauchy sequences with $O(\frac{1}{n})$ convergence, that is:

$$\exists B\, \forall m,\, n\, .\, |x_m - x_n| < B(\frac{1}{m} + \frac{1}{n})$$

Then he represents a rational sequence x_n by the natural number sequence a_n such that $x_n = \frac{a_n}{n}$. We can not do this for all rational sequences as some converge too quickly, for example $x_n = \frac{1}{2^n}$. However, as the rationals are dense in $\mathbf{R}$, we can choose another sequence representing the same real number; in the case of $x_n = \frac{1}{2^n}$ we could choose $x_n = \frac{1}{n}$ instead.

Now two rational sequences $x_n = \frac{a_n}{n}$ and $y_n = \frac{b_n}{n}$ represent the same real number if

$$\exists B\, \forall n\, .\, |a_n - b_n| \leq B$$

This gives the positive reals, which are then extended to the reals by representing a *signed real* x as a pair (x_1, x_2) of unsigned reals (positive reals as above) such that $x = x_1 - x_2$. Since each unsigned real has infinitely many representations, so does a signed real. Two reals $x = (x_1, x_2)$ and $y = (y_1, y_2)$ are then equal if, and only if, $x_1 + y_2 = y_1 + x_2$.

Harrison then defines the standard operations on the reals, for example $(x_1, x_2) * (y_1, y_2) = (x_1 * y_1 + x_2 * y_2, x_1 * y_2 + x_2 * y_1)$, and proves completeness of the reals, that is *every set of reals which is bounded above has a least upper bound.*

Convergence Nets Whereas Dutertre defined convergence and thus continuity using ε–δ definitions as seen in Chapter 4, Harrison's development in HOL Light takes a different approach to working with limits of functions and sequences. Harrison uses the notion of *convergence nets*, which allows an abstraction from both functions and sequences when considering convergence. By using this generalisation Harrison only needs to prove once that limits are unique, a sum of limits is the limit of the sum etc. Once all these proofs are done in the general setting of convergence nets, the theorems can easily be specialised to functions and sequences, by proving that the functions and sequences have the appropriate structures.

Division In HOL Light functions are total and there is no convenient method for defining subtypes. This led Harrison to define $0^{-1} = 0$ whereas in PVS 0^{-1} is undefined.

In PVS, division often causes the generation of TCCs similar to $x \neq 0$, but by using the sub-typing mechanisms in PVS many of these TCCs can be automatically discharged.

Having established limits for both functions and sequences Harrison goes on to develop series by defining finite sums, then to consider the convergence of sequences of these. He then proceeds to define differentiability and prove some classical theorems of real analysis, such as the mean and intermediate value theorems.

So the main differences between the PVS and HOL Light implementations of real analysis are in the implementation of reals, which in PVS are axiomatised, with part of the knowledge built into the decision procedures, and in HOL Light are constructed using a variation of Cantor's method. In HOL Light convergence is developed via the more general notion of convergence nets whereas in PVS ε–δ definitions are used. However, once past these foundational developments, in general the same theorems (although sometimes different variants) are proved in both PVS and HOL Light, so for the end user the two libraries look much the same.

5.2 Infinite Series

By definition the sum of an infinite series $\sum_{i=0}^{\infty} f(i)$ is the limit of the sequence $\sum_{i=0}^{n} f(i)$ as $n \to \infty$. If no such limit exists the series is said to be divergent. Therefore we first define the notion of partial sums, $\sum_{i=0}^{n} f(i)$.

In defining finite sums, Harrison used a somewhat unusual definition, namely

$$\texttt{sum(n, m, f)} = \sum_{i=n}^{n+m-1} f(i)$$

The reason for using this definition is that some proofs become easier, as one can often simply use induction in `m`. We initially used the same definition, but as it is not intuitive and reconstructing proofs based on mathematics books was more difficult, we changed to the common definition

$$\texttt{sum(n, m, f)} = \sum_{i=n}^{m} f(i)$$

By using the standard mathematical definition of finite sums it is more obvious that we are indeed proving standard theorems and it also turned out to be easier to use in the further development. Also, there were no significant complications in the proofs compared to when using Harrison's definition.

The PVS code defining sum is then:

```
sumc(n, m, f): RECURSIVE real =
  IF m < n THEN 0
  ELSE IF m = n THEN f(n)
       ELSE sumc(n, m - 1, f) + f(m)
       ENDIF
  ENDIF
 MEASURE m

sum(n, m)(f): real = sumc(n, m, f)
```

Using the new definition of finite sums we then proved many of the usual properties of finite sums, some of which are shown in Figure 5.1. The proofs are mostly by induction in the number of terms in the sum, and in general the proofs are straightforward, thus were done directly without referring to text books. The longest proof is that of sum_two which has 33 proof steps, although this could undoubtedly be shortened.

Next, we define infinite series. We say that a series converges if the sequence of its partial sums converges. In that case, the sum is the value of the limit of the sequence.

Definition 5.1 (Convergence of Series) *Let* $f : \mathrm{N} \to \mathrm{R}$, *then we say that the series*

$$\sum_{i=0}^{\infty} f(i)$$

converges if the sequence

$$a_n = \sum_{i=0}^{n} f(i)$$

converges. In that case, the value of the series is

$$\sum_{i=0}^{\infty} f(i) = \lim_{n \to \infty} a_n$$

```
sum_two: LEMMA FORALL f, n, m:
  sum(0, n)(f) + sum(n + 1, m)(f) = sum(0, max(m, n))(f)

abs_sum: LEMMA FORALL f, n, m: abs(sum(n, m)(f)) <= sum(n, m)(abs(f))

sum_lt: LEMMA FORALL f, g, n, m:
  (FORALL r: n <= r AND r <= n + m IMPLIES f(r) < g(r)) IMPLIES
    sum(n, n + m)(f) < sum(n, n + m)(g)

sum_eq: LEMMA FORALL f, g, n, m:
  (FORALL r: n <= r AND r <= n + m IMPLIES f(r) = g(r)) IMPLIES
    sum(n, n + m)(f) = sum(n, n + m)(g)

sum_add: LEMMA FORALL f, g, n, m:
  sum(n, m)(f + g) = sum(n, m)(f) + sum(n, m)(g)

sum_cmul: LEMMA FORALL f, c, n, m: sum(n, m)(c * f) = c * sum(n, m)(f)

sum_neg: LEMMA FORALL f, n, m: sum(n, m)(-f) = -sum(n, m)(f)

sum_group: LEMMA FORALL n, k, f:
  sum(0, n)(LAMBDA r: sum(r * (k + 1), (r + 1) * (k + 1) - 1)(f)) =
    sum(0, (n + 1) * (k + 1) - 1)(f)

sum_offset: LEMMA FORALL f, n, k:
  sum(0, n)(LAMBDA r: f(r + k + 1)) = sum(0, n + k + 1)(f) - sum(0, k)(f)

sum_reindex: LEMMA FORALL f, n, m, k:
  sum(n + k, m + k)(f) = sum(n, m)(LAMBDA r: f(r + k))

sum_0: LEMMA FORALL n, m: sum(n, m)(0) = 0

sum_cancel: LEMMA FORALL f, n, m:
  sum(n, m)(LAMBDA r: (f(r + 1) - f(r))) = f(max(m + 1, n))-f(n)
```

Figure 5.1: Lemmas about finite sums

In PVS we first define the notion of convergence of series, using the two argument version of convergence for sequences:

```
sums(f, s): bool
  = convergence(LAMBDA r: sum(0, r)(f), s)

summable(f): bool = EXISTS s: sums(f, s)
```

If a series $\sum_{i=0}^{\infty} f(i)$ is convergent then the sum can be extracted from the definition of sums using the *choice operator* `epsilon` of PVS:

```
suminf(f): real = epsilon(LAMBDA s: sums(f, s))
```

Here `epsilon` is used to extract *the* `s` such that `sums(f, s)` is true, that is to extract the value of a convergent series.

As for finite sums, we proved several theorems about arithmetic operations on series, some of which can be seen in Figure 5.2. Most of these theorems have standard proofs which can be found in different Mathematics Textbooks and replicated in PVS by filling in the details. Harrison used several lemmas in order to prove `seq_power`, and we used those lemmas as a guide for our proof.

With the foundations laid we can now go on to use these theorems. Without aid in determining convergence one would have to go back to the definition for each series. This would be an unreasonable burden to put on any user, so we develop the following three convergence criteria for series:

Theorem 5.1 (Cauchy Criteria for Series) *The series*

$$\sum_{i=0}^{\infty} f(i)$$

converges if and only if

$$\forall \varepsilon > 0 \, \exists N \in \mathbb{N} \, . \, \forall n > N \, . \, \forall m : \left| \sum_{i=n}^{n+m} f(i) \right| < \varepsilon$$

Remembering that `sum(n,m)(f)` is the finite sum over f starting at index n with m terms, we express the Cauchy criteria as follows:

```
ser_cauchy: LEMMA FORALL f: summable(f) =
  (FORALL e: EXISTS N: FORALL n, m: N <= n IMPLIES abs(sum(n, m)(f)) < e)
```

The proof of this theorem is based on that of [Mad91] and studies of Harrison's HOL Light proof script for the theorem.

```
sum_uniq2: LEMMA FORALL f, s, ss: sums(f, s) AND sums(f, ss) IMPLIES s = ss

ser_0: LEMMA FORALL f, n:
  (FORALL m: (n <= m) IMPLIES (f(m) = 0)) IMPLIES sums(f, sum(0, n)(f))

ser_group: LEMMA FORALL f, k: summable(f) AND 0 < k IMPLIES
  sums(LAMBDA n: sum(n * k, (n + 1) * k - 1)(f), suminf(f))

ser_pair: LEMMA FORALL f: summable(f) IMPLIES
  sums(LAMBDA n: sum(2 * n, 2 * n + 1)(f), suminf(f))

ser_offset: LEMMA FORALL f: summable(f) IMPLIES
  (FORALL k: sums(LAMBDA n: f(n + k + 1), suminf(f)-sum(0, k)(f)))

ser_pos_lt: LEMMA FORALL f, n: (summable(f) AND
  (FORALL m: n <= m IMPLIES (0 < f(m)))) IMPLIES sum(0, n)(f) < suminf(f)

ser_pos_eq: LEMMA FORALL f: summable(f) AND
  (FORALL m: 0 <= f(m)) AND 0 = suminf(f) IMPLIES (FORALL n: 0 = f(n))

ser_add: LEMMA FORALL f, f0, g, g0:
  sums(f, f0) AND sums(g, g0) IMPLIES sums(f + g, f0 + g0)

ser_cmul: LEMMA FORALL f, f0, c: sums(f, f0) IMPLIES sums(c * f, c * f0)

ser_neg: LEMMA FORALL f, f0: sums(f, f0) IMPLIES sums(-f, -f0)

ser_cdiv: LEMMA FORALL f, f0, c:
  sums(f, f0) AND c /= 0 IMPLIES sums(f / c, f0 / c)

ser_zero: LEMMA FORALL f: summable(f) IMPLIES convergence(f, 0)

ser_le: LEMMA FORALL f, g: (FORALL n: f(n) <= g(n)) AND summable(f) AND
  summable(g) IMPLIES suminf(f) <= suminf(g)

ser_abs: LEMMA FORALL f: summable(abs(f)) IMPLIES
  abs(suminf(f)) <= suminf(abs(f))

seq_power: LEMMA
  FORALL x0: abs(x0) < 1 IMPLIES convergence(LAMBDA n: x0 ^ n, 0)
```

Figure 5.2: Lemmas about infinite series

Theorem 5.2 (Comparison Criteria for Series) *Let $f, g : \mathbf{N} \to \mathbf{R}_+$. If the series*

$$\sum_{i=0}^{\infty} g(i)$$

converges and

$$\exists c > 0\, N \in \mathbf{N}\, \forall n > N\, .\, f(n) < cg(n)$$

Then the series

$$\sum_{i=0}^{\infty} f(i)$$

also converges.

In PVS, we prove a slightly different version, where we assume $c = 1$ and allow f to take non-positive values:

```
ser_compar: LEMMA FORALL f, g:
  (EXISTS N: FORALL n: N <= n IMPLIES abs(f(n)) <= g(n)) AND summable(g)
    IMPLIES summable(f)
```

The theorem we prove in PVS is in fact more general than Theorem 5.2, since the function f may take non-positive values. It is also the case that if $\sum_{i=0}^{\infty} g(i)$ converges, then so does $\sum_{i=0}^{\infty} \frac{g(i)}{c}$ for any non-zero c and $\frac{g}{c}$ may then be used in Theorem 5.2. Thus `ser_compar` implies Theorem 5.2. The proof of `ser_compar` is based on those of [Mad91, Bra90].

Finally, we consider the ratio test for convergence:

Theorem 5.3 (Ratio Criteria for Series) *Let $f : \mathbf{N} \to \mathbf{R}_+$. Then*

$$\sum_{i=0}^{\infty} f(i)$$

converges if

$$\exists c \in (0,1)\, N \in \mathbf{N}\, \forall n > N\, .\, \frac{f(n+1)}{f(n)} < c$$

Again, the version we use in PVS is slightly modified, in that we allow f to take negative values, as long as the criteria is met on the absolute value of f:

```
ser_ratio_test: LEMMA FORALL f:
  (EXISTS c, N: c < 1 AND
    (FORALL n: N <= n IMPLIES abs(f(n + 1)) <= c * abs(f(n))))
      IMPLIES summable(f)
```

The proof of `ser_ratio_test` is based on an idea from [Bea97], where no detail is given, and on [Mad91].

Power series are a particular kind of series of the form $\sum_{i=0}^{\infty} f(i) * (k - x_0)^i$, where x_0 is the centre of the interval of convergence. We do not work specifically with the interval of convergence, however many theorems (such as `powser_insidee` below) have as a prerequisite that the power series is convergent for some k, and then use $|k|$ instead of the radius of convergence as $|k|$ is less than or equal to the radius of convergence. Thus our prerequisites are in general sufficient, but not necessary. In our development we only deal with $x_0 = 0$ although some proofs use different centres. Due to their regular form, power series are particularly well-behaved and so we prove two convergence criteria specific to power series.

Theorem 5.4 (Convergence of Geometric Series) *If* $|x| < 1$ *then the series*

$$\sum_{i=0}^{\infty} x^i$$

converges with sum $\frac{1}{1-x}$.

In PVS, this is expressed as follows:

```
gp: LEMMA
  FORALL x: abs(x) < 1 IMPLIES sums(LAMBDA r: x ^ r, 1 / (1 - x))
```

The proof of gp relies on several lemmas and is mainly based on Harrison's lemmas and proof scripts. However, we later found a very elegant and short proof in [Apo74]. This uses the fact that by pairing up terms we get

$$\forall n \in \mathbb{N} : (1-x)\sum_{i=0}^{n} x^i = \sum_{i=0}^{n} x^i - x^{i+1} = 1 - x^{n+1}$$

Since for $|x| < 1$ we have $\lim_{n\to\infty} x^{n+1} = 0$, the proof is completed.

As power series have an interval of convergence, we know that strictly inside this interval, the series is convergent.

Theorem 5.5 (Interval of Convergence for Power Series) *If*

$$\sum_{i=0}^{\infty} f(i)k^i$$

converges and $|x| < |k|$ *then the series*

$$\sum_{i=0}^{\infty} f(i)x^i$$

also converges.

In PVS, this is expressed as follows:

```
powser_insidee: LEMMA FORALL f, k, x:
  summable(LAMBDA n: f(n) * (k ^ n)) AND abs(x) < abs(k)
    IMPLIES summable(LAMBDA n: f(n) * (x ^ n))
```

We also proved a theorem about differentiation of power series. Harrison proved that if a power series

$$F(k) = \sum_{i=0}^{\infty} f(i) * k^i \tag{5.1}$$

and the power series obtained by differentiating termwise once and twice

$$G(k) = \sum_{i=1}^{\infty} if(i) * k^{i-1}$$

$$H(k) = \sum_{i=2}^{\infty} i(i-1)f(i) * k^{i-2}$$

all are convergent then for $x \in \mathbf{R}$ with $|x| < |k|$ the derivative of the power series $F(x)$ is $G(x)$. Here $|k|$ is used instead of the radius of convergence as explained above. Whenever we wish to use this theorem we then need to prove that the first and second termwise derivatives exist. When F is an elementary function, G and H are often identical to known series, so we know that they are convergent and hence can use the theorem to prove that G is the derivative of F.

The usual proof of termwise differentiation of power series relies on termwise integration of power series. The series obtained by termwise integration of a power series is again a power series and the two series have the same interval of convergence. Thus if the series

$$\sum_{i=0}^{\infty} f(i) * y^i$$

is convergent on $(-\rho, \rho)$, we know that

$$\forall x \in (-\rho, \rho) \; : \; \int_0^x \sum_{i=0}^{\infty} f(i) * y^i \, dy = \sum_{i=0}^{\infty} \int_0^x f(i) * y^i \, dy = \sum_{i=0}^{\infty} \frac{f(i)}{i+1} x^{i+1}$$

This can then be used to prove the theorem about termwise differentiation of power series. However, as we have not developed support for reasoning about integration in PVS, we can not use this proof.

The actual proof we used is quite complicated. It is based purely on Harrison's lemmas and HOL Light proof scripts.

We also proved a more general theorem:

Theorem 5.6 (Differentiation of Power Series) *If* $\sum_{i=0}^{\infty} f(i) * k^i$ *converges then*

$$\forall x \, . \, |x| < |k| \Rightarrow (\sum_{i=0}^{\infty} f(i) * x^i)' = \sum_{i=0}^{\infty} (i+1) f(i+1) * x^i$$

We did this by first proving the same theorem as Harrison and then proving that for all absolutely convergent power series the power series obtained by termwise differentiation is convergent. The idea of this proof is to use comparison between the series

$$\sum_{i=1}^{\infty} i|f(i)| * |x|^{i-1} \tag{5.2}$$

and

$$\sum_{i=2}^{\infty} |f(i)| * \frac{(d+|x|)^i}{d} \tag{5.3}$$

where d is a positive real so that $d + |x| < |k|$, which implies that

$$\sum_{i=0}^{\infty} |f(i)| * (d+|x|)^i$$

is convergent. We prove that for $i \geq 2$ we have $i|f(i)| * |x|^{i-1} \leq |f(i)| * \frac{(d+|x|)^i}{d}$ and since (5.3) converges (5.2) also converges. The suggestion to use this particular comparison comes from [Bea97], although not much detail is given there.

5.3 Some Transcendental Functions

As we now have the tools to handle specific power series, we can define some transcendental functions. We start by defining exp and its inverse ln, then we go on to define cos and sin and functions related to them. This also leads us to a definition of π.

5.3.1 Exp and ln

The function exp is defined by the following power series:

$$\exp(x) = \sum_{i=0}^{\infty} \frac{1}{i!} x^i$$

In PVS we first define the finite sum

```
sum(0, n)(LAMBDA m: exp_ser(m) * (x ^ m))
```

where

```
exp_ser(m) = 1 / fac(n)
```

Then we prove that the sequence s_n with

$$s_n = \sum_{i=0}^{n} \frac{1}{i!} x^i$$

is convergent using the ratio test [Mad91] and finally we can define the function $\exp : \mathbf{R} \to \mathbf{R}$ by

```
exp(x): real = suminf(LAMBDA n: exp_ser(n) * (x ^ n))
```

We can now prove that exp is differentiable with derivative exp using Theorem 5.6. As we know that differentiability implies continuity, we now know that exp is also continuous.

For each of the functions described in this section, there is a large collection of well-known facts like those that can be found in many text books. We have proved a useful collection of these. Firstly about exp:

$$\exp(0) = 1$$

$$\forall x \, \forall y \, . \, \exp(x + y) = \exp(x) \exp(y)$$

$$\forall x \, \forall y \, . \, \exp(x - y) = \frac{\exp(x)}{\exp(y)}$$

We can define ln as the inverse of exp, but to do so we must first prove that such an inverse does indeed exist. The following theorem asserts that exp is surjective on $\mathbf{R}_+$:

$$\forall y \, . \, 0 < y \Rightarrow \exists x \, . \, \exp(x) = y, \text{ where } x \text{ is unique} \tag{5.4}$$

Then ln is defined on the positive reals by using the choice operator `epsilon` in PVS

```
log(x): real = epsilon(LAMBDA y: exp(y) = x)
```

So $\ln(x)$ gives a y such that $\exp(y) = x$, and we know from (5.4) that this y is unique.

There are a few theorems about ln:

$$\forall x \, . \, \ln(\exp(x)) = x$$

$$\ln(1) = 0$$

$$\forall x > 0 \, \forall y > 0 \, . \, \ln\left(\frac{x}{y}\right) = \ln(x) - \ln(y)$$

5.3.2 Cosine and Sine

We now define cos and sin by their power series. To prove convergence of these series we used the comparison test [Mad91] and the fact that the power series for exp is convergent.

cos is defined in the following way:

$$\cos(x) = \sum_{i=0}^{\infty} \frac{(-1)^i}{(2i)!} x^{(2i)}$$

And sin is defined in the following way:

$$\sin(x) = \sum_{i=0}^{\infty} \frac{(-1)^i}{(2i+1)!} x^{(2i+1)}$$

Again we use Theorem 5.6 to prove that $(-\sin)$ is the derivative of cos and cos is the derivative of sin:

$$\cos' = -\sin$$

$$\sin' = \cos$$

We proved a large collection of standard facts about cos and sin. Here are some of the theorems:

$$\cos(0) = 1$$

$$\sin(0) = 0$$

$$\forall x \,.\, \sin(x)^2 + \cos(x)^2 = 1$$

$$\forall x \,.\, -1 \leq \cos(x) \leq 1$$

$$\forall x \,.\, -1 \leq \sin(x) \leq 1$$

$$\forall x \,\forall y \,.\, \sin(x+y) = \sin(x)\cos(y) + \cos(x)\sin(y)$$

$$\forall x \,.\, \sin(-x) = -\sin(x)$$

We now define π. This can be done in several different ways, but we chose to do as Harrison and define π as *two times the first positive zero of cosine.* So first we need to prove that there is such a zero, in fact we prove that it exists in $[0, 2]$ and is unique:

$$\exists x \,.\, 0 \leq x \leq 2 \;\wedge\; \cos(x) = 0 \;\wedge\; (\forall y \,.\, (0 \leq y \leq 2 \;\wedge\; \cos(y) = 0) \;\Rightarrow\; y = x) \qquad (5.5)$$

The proof is done by first proving that $\cos(2) < 0$, this is done by grouping terms in the power series, offsetting the power series and then grouping the terms in the resulting power series. Then we use the intermediate value theorem (which Dutertre included in the real analysis library), so we know that there is a value x between 0 and 2 for which $\cos(x) = 0$ as cosine is continuous and $\cos(0) = 1$. The uniqueness is proven by assuming that $\cos(x) =$

0 and $\cos(y) = 0$. As suggested by [Mad91] we then consider $\sin(|x - y|)$ which can be expressed using $\sin(x)$, $\sin(y)$, $\cos(x)$ and $\cos(y)$, proving that $\sin(|x - y|) = 0$. However, we also know that sin is positive on the interval $(0, 2)$ and thus reach the conclusion that $x = y$.

Finally we use the choice operator to extract the x found in (5.5) and define π as $2x$. The definition then immediately tells us that $0 < \pi < 4$. These are obviously not very good bounds, and so we prove that following result:

$$3 < \pi < 3.2$$

This proof is similar to that of (5.5) in that it relies on first proving that $\cos(1.5) > 0$ and $\cos(1.6) < 0$ (by grouping, offsetting and re-grouping the power series) and then using the intermediate value theorem to find a zero for cosine between 1.5 and 1.6. Finally we note that as the zero in (5.5) is unique, $\frac{\pi}{2}$ must be between 1.5 and 1.6. In fact it was done by following a suggested proof from [Apo74]: proving that $\sqrt{2} < 1.5$, $1.6 < \sqrt{3}$, $\cos(\sqrt{2}) > 0$ and $\cos(\sqrt{3}) < 0$ and using the intermediate value theorem on this result. We decided to generalise this result in order to make it a bit easier to develop yet closer bounds should we ever need to. The following theorem tells us how values between $\sqrt{2}$ and $\sqrt{3}$ relate to $\frac{\pi}{2}$:

Theorem 5.7 (Bounds on π)

$$\begin{aligned}\forall x \,.\, \sqrt{2} \leq x \leq \sqrt{3} \Rightarrow ((\cos(x) < 0 \Rightarrow \pi < 2x) \\ \wedge (\cos(x) > 0 \Rightarrow \pi > 2x) \\ \wedge (\cos(x) = 0 \Rightarrow \pi = 2x))\end{aligned}$$

So if we ever wish to tighten the bounds on π we do not need to use the intermediate value theorem or anything similar to that, all we need to do is determine the value of cosine on the bounds, i.e. $\forall x, z \in [\sqrt{2}, \sqrt{3}] \,.\, \cos(y) < 0 < \cos(x) \Rightarrow 2x < \pi < 2y$

We now present various lemmas relating cos, sin and π:

$$\pi > 0$$

$$\cos(\pi) = -1$$

$$\sin(\pi) = 0$$

$$\forall x \,.\, \sin(x) = \cos(\frac{\pi}{2} - x)$$

$$\forall n \,.\, \sin(n\pi) = 0$$

The next two theorems state exactly where the zeros of cos and sin are:

Theorem 5.8

$$\forall x \,.\, \cos(x) = 0 \Leftrightarrow ((\exists\, n \,.\, x = (2n+1)\frac{\pi}{2}) \vee (\exists\, n \,.\, x = -(2n+1)\frac{\pi}{2}))$$

Theorem 5.9

$$\forall x \,.\, \sin(x) = 0 \Leftrightarrow ((\exists\, n \,.\, x = n\pi) \vee (\exists\, n \,.\, x = -n\pi))$$

PROOF: In order to prove Theorem 5.8 about the zeros of cos we first need another three lemmas. We simply follow Harrison's development of lemmas without considering the proof scripts.

Lemma 5.1 $\cos(-x) = \cos(x)$

Lemma 5.2 $\cos((2n+1)\frac{\pi}{2}) = 0$

Lemma 5.3 $(x \geq 0 \wedge \cos(x) = 0) \Rightarrow \exists\, n \,.\, x = n\frac{\pi}{2} \wedge \mathit{odd}(n)$

Now we can prove Theorem 5.8 by considering the following cases:

$\Rightarrow$	$x \geq 0$:	use Lemma 5.3
	$x < 0$:	use Lemma 5.1, then Lemma 5.3
$\Leftarrow$	$x = (2n+1)\frac{\pi}{2}$:	use Lemma 5.2
	$x = -(2n+1)\frac{\pi}{2}$:	use Lemma 5.1, then Lemma 5.2

So given the three lemmas, this proof is quite straightforward. However, note that in the second half of the proof we distinguish between positive and negative x, this is to support the induction necessary to complete the proof.

The proof of Lemma 5.1 uses various manipulations of cos and sin, and eventually comes to proving that $(\sin(-x)+\sin(x))^2+(\cos(-x)-\cos(x))^2$ is constant zero. This is essentially

done by first proving that $(\sin(-x)+\sin(x))^2+(\cos(-x)-\cos(x))^2$ differentiates to zero everywhere, and so is continuous and thus is a constant function, and then observing that it is zero at zero. The proof of Lemma 5.2 is by induction on n, using Lemma 5.1 and the fact that $\cos(x+\pi) = -\cos(x)$. The proof of Lemma 5.3 uses the other lemmas. The essential part here is that cos has an inverse on the interval $[0,\pi]$. ■

5.3.3 Tangent

The function $\tan(x)$ is undefined for values of x where $\cos(x) = 0$. As PVS is strongly typed we need to work out the domain for tan, that is $\{x \in \mathbf{R} | \cos(x) \neq 0\}$. In the previous section we saw that $\cos(x) = 0$ exactly when $x = k\frac{\pi}{2}$ for some odd k, and we use this in the representation of the type for the domain of tan:

```
x: VAR real
k: VAR int
cos_nz_type: NONEMPTY_TYPE
  = { x | FORALL k: x /= (2 * k + 1) * pi / 2 }
```

The type itself is not related to cos yet, so we use a judgement to state (and prove) that cos applied to any element in cos_nz_type is indeed non-zero:

```
cos_nz_nz: JUDGEMENT cos(x_cos_nz) HAS_TYPE nzreal
```

We also tried the more direct definition

```
cos_nz_type: NONEMPTY_TYPE = { x | cos(x) /= 0 }
```

but this was no easier to work with, and so we reverted to our original definition above.

We can now define tan and prove some lemmas about it, using manipulations on the definition:

$$\forall x \,.\, \cos(x) \neq 0 \;\Rightarrow\; \tan(x) = \frac{\sin(x)}{\cos(x)}$$

$$\tan(n\pi) = 0$$

$$\forall x \,.\, \cos(x) \neq 0 \;\Rightarrow\; \tan(-x) = -\tan(x)$$

The next lemma is about tan of a sum. The requirement $\cos(x), \cos(y), \cos(x+y) \neq 0$ ensures not only that tan is defined but also that $\tan(x)\tan(y) \neq 1$, thus the fraction is well-defined.

$$\forall x\, \forall y \,.\, (\cos(x), \cos(y), \cos(x+y) \neq 0) \;\Rightarrow\; \tan(x+y) = \frac{\tan(x)+\tan(y)}{1-\tan(x)\tan(y)}$$

Finally we prove that tan has an inverse on the open interval $(-\frac{\pi}{2}, \frac{\pi}{2})$:

$$\forall y\, \exists x \,.\, \cos(x) \neq 0 \;\wedge\; -\frac{\pi}{2} < x < \frac{\pi}{2} \;\wedge\; \tan(x) = y \;\wedge\; x \text{ is unique}$$

5.4 Continuity and Differentiation

In the previous section we saw that the proof of Theorem 5.8 relies on differentiating functions involving cos and sin. Here we use Theorem 5.6 about termwise differentiation of power series to differentiate exp, cos and sin, and get the following results:

$$\exp'(x) = \exp(x)$$

$$\cos'(x) = -\sin(x)$$

$$\sin'(x) = \cos(x)$$

Dutertre proved that if a function is differentiable it is also continuous, so we use the results above to prove that each of the functions exp, cos and sin are continuous everywhere.

Now given derivatives for cos and sin we can use the general theorems about function combinations preserving differentiability to prove that tan is differentiable and find its derivative. Dutertre's implementation of differentiation requires the domain considered to be connected, which of course the domain of tan is not, as tan is undefined at odd multiples of $\frac{\pi}{2}$. However we know and prove in PVS that $\forall x \in \mathit{domain}(\tan)$ there is an interval $I_k = (k\pi - \frac{\pi}{2}, k\pi + \frac{\pi}{2})$ of length π containing x such that $I_k \subset \mathit{domain}(\tan)$ and I_k is connected, as shown in Figure 5.3. Thus by restricting tan to I_k we can differentiate tan on I_k and we get $\tan'(x) = \frac{1}{\cos(x)^2}$. From this also follows that tan is continuous everywhere on its domain.

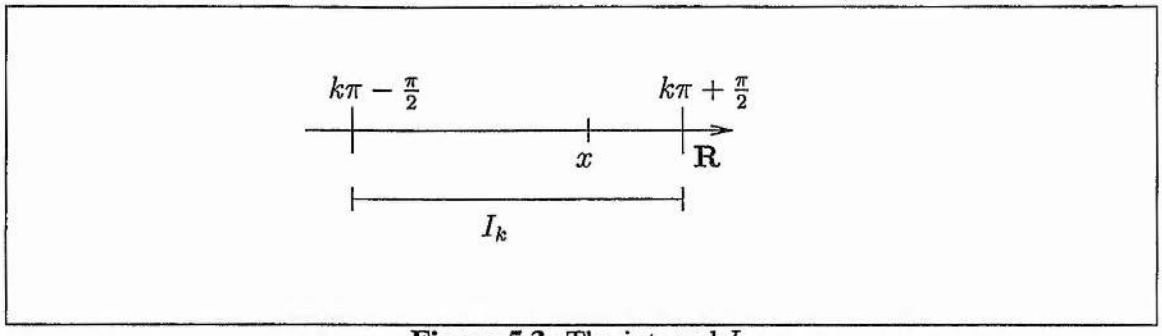


Figure 5.3: The interval I_k

5.4.1 Example Proof

The proofs in this development seems to fall into two categories, either they are small and quite easy, or they are more complicated and tend to get very long. Proofs about properties of finite series fall into the first category, as they tend to be simple induction proofs. As one might expect, a great number of the theorems about properties of the trigonometric functions are quite simple. This is because once the basic tools are in place we no longer have to go back to the ε–δ definitions to prove convergence to certain values. Between these two extremes of the development are a number of theorems which in general require a lot of work.

The harder proofs were about infinite series and power series. It is interesting here to note that these proofs are also difficult to do by hand, particularly since most textbooks leave out many details that "are obvious" yet require several non-trivial proof steps to establish formally. So apart from the added difficulty that comes from doing the proofs formally, the level of difficulty was much the same using PVS as it is to do the proofs by hand. However, one theorem in particular stands out from this generalisation: the one about differentiation of power series (Theorem 5.6). In most textbooks it is proven by first introducing integration, then proving that an absolutely convergent power series can be integrated termwise and then applying this to the power series obtained by doing termwise differentiation of the original series. However, as we have not implemented integration, this was not a feasible proof for us and a more complicated proof was necessary as described in Section 5.2. Finding suitable bounds for π took some time too. This was not actually because of the PVS proof being complicated, but because finding values that made the proofs of the sign of the values of cosine easier took some effort. However, once the proofs were done on paper they translated directly into PVS.

In general, we used various textbooks [Apo74, Bea97, Bra90, TF88, Mad91, Kap57] as guides for the proofs. However the textbook proofs were not always feasible for a variety of reasons. We have already seen that they may depend on theory not implemented in PVS, such as integration. Another problem we encountered was with geometrical reasoning which could not easily be formalised.

Let us now consider one of the easier proofs. We would like to prove the following theorem:

```
sin_cos: THEOREM FORALL x: sin(x) = cos(pi / 2 - x)
```

These four lemmas are used in the proof:

```
cos_add: LEMMA
  FORALL x, y: cos(x + y) = cos(x) * cos(y) - sin(x) * sin(y)

cos_pi2: LEMMA cos(pi / 2) = 0

sin_pi2: LEMMA sin(pi / 2) = 1

sin_neg: LEMMA FORALL x: sin(-x) = -sin(x)
```

The proof is as follows:

```
(SKOLEM!)
(USE "cos_add" ("x" "pi/2" "y" "-x!1"))
(LEMMA "cos_pi2")
(LEMMA "sin_pi2")
(USE "sin_neg" ("x" "x!1"))
(ASSERT)
```

It applies lemma `cos_add` to `pi / 2` and `-x` to get

```
cos(pi / 2 - x) = cos(pi / 2) * cos(-x) - sin(pi / 2) * sin(-x)
```

Then we use lemma `cos_pi2` and `sin_pi2` to get

```
cos(pi / 2 - x) = 0 * cos(-x) - 1 * sin(-x)
```

And finally using `sin_neg` we get

```
sin(-x) = -sin(x)
```

And this completes the proof.

5.5 Comparison with a Trigonometric Function Library

Researchers at ICASE and NASA Langley have developed a PVS library of trigonometric functions [MCB+]. Their approach, however, is very different to ours in that they use an axiomatisation of the trigonometric functions. Based on these definitions and axioms, they prove many of the identities we also proved. They also include a two-argument version of arctan, some geometric observations on cos, and translations between degrees and radians.

There are two main differences between the ICASE/NASA Langley library of trigonometric functions and our own library:

Approximation In the ICASE/NASA Langley library, it is proven that the definitions of $\cos(x)$, $\sin(x)$ and $\tan(x)$ are within certain bounds given by the power series, for instance about $\sin(x)$: the sum of the first 4 terms in the power series for sin is a lower bound, and the sum of the first 5 terms in the power series for sin is an upper bound. This is, of course, done without defining power series. They also prove various (in)equalities for the approximations of each of $\cos(x)$, $\sin(x)$ and $\tan(x)$. Since calculating the first few terms of any of the power series involved is quite easy, this provides a means to work with values albeit in the form of approximations.

Analysis Since the library was developed mainly to work within discrete mathematics, there is no support for real analysis. This in particular means that for example continuity checking would not be possible without first either developing large amounts of analysis or importing a corrected version of Dutertre's library.

The library is being used in applications such as AILS (see Section 2.3.3), and it would be

interesting to either combine these two libraries, or to add a theory of approximation to our own library in order to prove more about values.

5.6 Summary

In this chapter we have seen an overview of our implementation of elementary functions. The implementation uses Dutertre's real analysis library (Chapter 4) as a basis and roughly follows the HOL Light development by Harrison [Har98]. In Section 5.1.1 we described the main differences between the PVS and HOL Light implementations of real numbers and convergence of functions. Whereas the theory and techniques behind these are different in the two systems, when it came to actually using the PVS implementation it was not a problem as largely the same main theorems are proven in both implementations.

We have described how we implemented transcendental functions via power series, in particular using general theorems about finite sums, infinite series and power series. Harrison proved a result about differentiation of power series, namely that if both the first and second termwise derivatives of a power series exist and are convergent then the power series is differentiable with the first termwise derivative as its derivative. As we are concerned mainly with the power series defining exp, cos and sin and they differentiate termwise to each other, using Harrison's theorem is not a problem as the preconditions are easy to prove. However, we also proved that for an absolutely convergent power series the first termwise derivative does exist, and so we could prove the usual theorem about differentiation of power series, Theorem 5.6. Whereas this was not necessary it substantially shortened the proofs of differentiability of exp, cos and sin.

We have defined functions exp, ln, cos, arccos, sin, arcsin, and tan, and proved a large collection of facts about these. We have also defined π, again using the same method as Harrison, that is to define π as *"two times the first positive zero of cosine"*. We then proved that $3 < \pi < 3.2$, which is a close enough bound for all the problems we have been trying to solve so far.

Without doubt the hardest and longest proof in this implementation was that about differentiability of power series, it uses 7 substantial lemmas and combined they have about 1600 proof steps. However, one must also remember that this was done when we first started

working with PVS and with the experience we have now, we believe it is likely that the proofs could be shortened quite a bit in terms of number of proof steps, although it might require a considerable effort.

Our library is more extensive than that of ICASE/NASA Langley, since it supports not only the functions exp and ln in addition to the trigonometric function, but also various analytical properties. However, the approximations introduced in the ICASE/NASA Langley library are very useful, and we believe that a combination of the two libraries would have even more application areas than any of the libraries on their own.

Chapter 6

Checking Analytic Properties Automatically

In this chapter we describe how we can use PVS and the library described in Chapter 5 to check *automatically* certain analytic properties such as continuity and existence of limits. For simple functions, humans do this efficiently by just observing if the function is a well-formed combination of, say, continuous functions, and applying the knowledge that the continuity is then preserved. For more complicated functions, however, this is not so simple.

Our aim is to enable automatic checking of the analytic properties for a useful set of real-valued functions. We explore two different techniques for implementing automation in PVS. Method 1 is based purely on a collection of strategies which explicitly apply the theorems of Dutertre's `continuous_functions` (Figure 4.6). Method 2 uses type judgements extensively and then has a very simple top-level strategy performing some preprocessing before applying the PVS strategy `grind`. Our implementation of Method 1 only handles rational functions, and although it could be extended to cover elementary functions too, we did not do that as Method 2 works better and providing checking for elementary functions in Method 2 was quite easy. Each method is implemented with a top-level strategy which is called without arguments for ease of use.

Section 6.1 briefly describes our test files, which contain examples of theorems proven using our automation. Section 6.2 gives an introduction to continuity checking. Section 6.3

describes the implementation of Method 1 and Section 6.4 describes the implementation of Method 2 and compares the two methods. In Section 6.5 we see how the judgements can be used to check other analytic properties too. All of our strategies can be found in Appendix A. Appendix B contains our files of examples of functions on which we have run Method 1 and/or Method 2, and the results of those tests. The nature of the examples is also discussed in Section 6.1. Appendix C contains an overview of all our strategies, and they are all included in the file `pvs-strategies` on the CD-ROM.

In general the question: "*given* $A \subseteq \mathbf{R}$, $f : A \to \mathbf{R}$ *and* $a \in A$ *is* f *continuous at* a*?*" is undecidable [Che80]. However various restrictions on f and A allow us to determine continuity and other analytic properties for a large class of generally useful and interesting functions.

Our implementation of Method 1 relies on the question to be proven by PVS being of one of the following types:

$$\text{domain:} \quad \forall x \,.\, \mathit{continuous}(f, x) \tag{6.1}$$

$$\text{point:} \quad \mathit{continuous}(f, a) \tag{6.2}$$

$$\text{precondition:} \quad \forall x \,.\, P(x) \Rightarrow Q(x) \tag{6.3}$$

where *domain* is for checking if a function is continuous on the whole or part of it domain, *point* is for checking if a function is continuous at a specific point and finally *precondition* is used for functions such as

$$f(x) = \frac{3}{a + x}$$

We can then check the lemma

$$\forall x > 0 \,.\, a \geq 0 \Rightarrow \mathit{continuous}(f, x)$$

whereas without the precondition $a \geq 0$ this would not in general be true. Initially Method 1 was constructed to handle only the *domain* case, however we extended it to handle *point* and *precondition* cases too. It is possible to abstract from these types and indeed we do that in Method 2, however as mentioned before Method 2 in general works better that Method 1, so we did not do the work to make this abstraction in Method 1.

6.1 Example files

Appendix B contains our test files, containing a total of 88 lemmas about continuity and convergence of functions, for example in Appendix B.1:

```
example1: LEMMA
  continuous[posreal](LAMBDA (x: posreal): x + 2 / x, 3)
```

That is, the function $f(x) = \frac{x+2}{x}$ defined on the positive real numbers is continuous at $x = 3$.

These test files have largely been constructed during the development of the strategies and judgements, but also later on by inclusion of functions suggested in discussions, for instance lemmas `example1` and `example2` in `differentialequation_examples` (Appendix B.4) arose when working with differential equations in [ADG+01].

For each example file there is also a table showing the type (domain, point or precondition as above) of the lemma and how well the different methods for continuity (or convergence) checking worked in each case. We tested the following methods for lemmas about continuity: Method 1, Method 1 with judgements and Method 2. For lemmas about convergence we tested Method 2. The result of each test is shown as either the *run time* as measured by PVS in seconds, an *error*, or *n/a* to indicate that the method does not apply to that lemma (for example Method 1 was never extended to handle transcendental functions). The measured run time is heavily dependent on other processes running on the machine at the time, and so should not be used as a basis for bold claims about effectiveness about one method over the other. In general though, for simpler examples, Method 1 seems faster, whereas for more complicated examples, Method 2 is faster. Some examples have quite high run times, this is often due to PVS doing garbage collection during the proof. Each proof which contains a garbage collection was re-run in order to try to avoid measuring the time of the garbage collection too. If this was successful, the lower time was given in the table. When applying Method 1, we do get the occasional error, this is simply due to Method 1 not being fully developed.

6.2 Continuity

We use Dutertre's [Dut96] `continuous_functions` theory as the basis for our implementation. It builds on the well-known definition of continuity,

Definition 6.1 *Let $A \subseteq \mathbf{R}$, $f : A \to \mathbf{R}$ and $a \in A$. We say that f is* continuous *at a if*

$$\forall \varepsilon \in \mathbf{R}_+ \, \exists \delta \in \mathbf{R}_+ \, \forall x \in A \, . \, |x - a| < \delta \Rightarrow |f(x) - f(a)| < \varepsilon$$

There are several different approaches to checking if a function is continuous at a certain point:

Direct: We can use the above definition directly by working out a δ given some ε, for example if $f = g_1 + g_2$ and g_1 and g_2 are both continuous at some a then by using the definition on both g_1 and g_2 (with $\varepsilon/2$) we get δ_1 and δ_2. Using the minimum of those δs and the definition of f we now see that f is also continuous at a.

Via Limits: An equivalent definition of continuity is using limits: f is continuous at $a \Leftrightarrow \lim_{x \in A \to a} f(x) = f(a)$, that is the limit exists and has the required value.

High School Method: By applying simple restrictions to the functions we can use a very simple method, which we call the *high school method.* It uses the fact that the identity and constant functions are continuous and that well-formed application of the operations addition, subtraction, multiplication, division, composition and absolute value preserves continuity.

Other Methods: It is well-known that if a function is differentiable (at a point) then it is also continuous (at that point). This can be used to prove continuity of functions where differentiability is easier to establish, in fact we used this method to prove exp, cos and sin continuous. Also, L'Hôpital's rule can be used when taking limits of fractions where both divisor and denominator are zero at the point of interest (see Example 6.2).

We have chosen to base our continuity checker on the high school method. Although it is a conceptually simple method it still solves many of the problems arising in the use of CAS. The method is syntax-directed in that by inspecting the expression describing the function we decide which action to perform next.

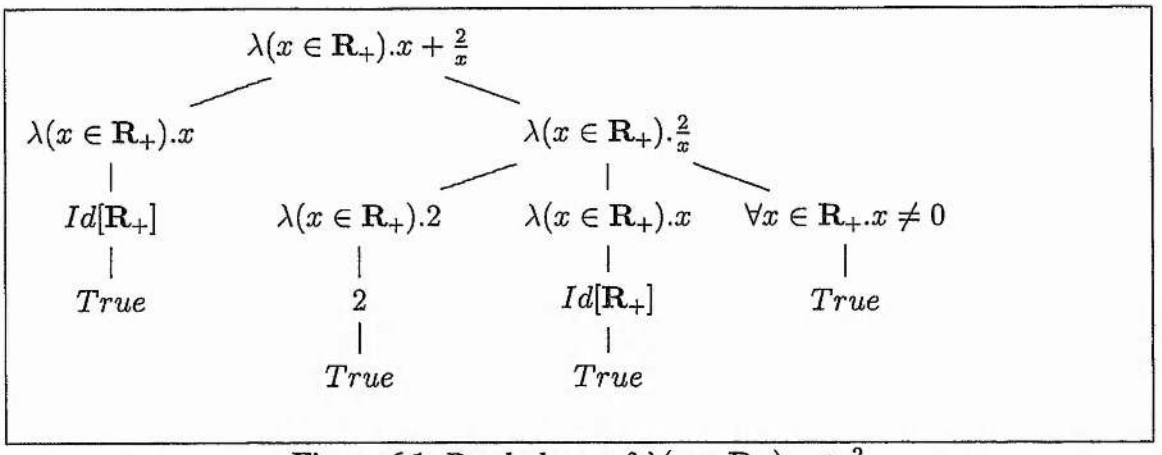


Figure 6.1: Break-down of $\lambda(x \in \mathbf{R}_+).x + \frac{2}{x}$

Example 6.1 *Consider the function $\lambda x\,.\,x + \frac{2}{x}$ defined for $x \in \mathbf{R}_+$. The outermost operator here is $+$, so first we use the theorem* `sum_continuous` *saying that the sum of two continuous functions is continuous. We then know that the function is continuous if each of the functions $\lambda x\,.\,x$ and $\lambda x\,.\,\frac{2}{x}$ are continuous. Likewise then for $\lambda x\,.\,\frac{2}{x}$; it is continuous if $\lambda x\,.\,x$ and $\lambda x\,.\,2$ are continuous and the division is well-formed, that is $\forall x \in \mathbf{R}_+\,.\,x \neq 0$ which is true. The base cases here are then $\lambda x\,.\,x$ and $\lambda x\,.\,2$, but as we know that the identity function and constant functions are continuous, the proof is finished. Figure 6.1 shows the steps of the proof.* □

What is described here is clearly a very basic method, and by no means a complete one. Any function defined by cases is outside the scope of this method, but let us consider an example, using L'Hôpital's rule.

Theorem 6.1 (L'Hôpital's Rule) *Given functions $N, D : (a, b) \to \mathbf{R}$ for $a, b \in \mathbf{R}$ where N and D are differentiable and both derivatives are continuous, and with*

$$N(x) = 0$$
$$D(x) = 0$$
$$D'(x) \neq 0$$

for some $x \in (a, b)$.

Then $\frac{N(y)}{D(y)}$ has a limit as y tends to x and the limit is $\frac{N'(x)}{D'(x)}$.

There is also a more general version of L'Hôpital's rule which deals with the case when the n first derivatives of both numerator and denominator are zero.

Example 6.2 *Consider the function*

$$f(x) = \frac{1 - \cos(x)}{\exp(x) - 1} \tag{6.4}$$

defined on $\mathbf{R} \setminus \{0\}$. *We see that at zero both the numerator* $N(x) = 1 - \cos(x)$ *and the denominator* $D(x) = \exp(x) - 1$ *are zero leaving* f *undefined at zero. However, using L'Hôpital's rule we can still determine if* f *has a limit at zero. First, we calculate the first derivatives of* N *and* D*:*

$$N'(x) = \sin(x)$$
$$D'(x) = \exp(x)$$

Then we see that $D'(0) = 1$ *and L'Hôpital's rule tells us that the limit of* f *at zero exists and is* $\frac{N'(0)}{D'(0)} = 0$.

Now if we define a new function

$$g(x) = \begin{cases} f(x) & \text{if } x \neq 0, \\ 1 & \text{if } x = 0. \end{cases} \tag{6.5}$$

then g *is defined on* $\mathbf{R}$ *and is continuous everywhere, although as* g *is defined by cases and relies on L'Hôpital's rule to determine the limit (and thus continuity) at zero it does not belong to the class of functions our automation in PVS can handle.* □

6.2.1 Automatic Continuity Checking

We wish to extend the PVS theory `continuous_functions` (see Figure 4.6) by Dutertre [Dut96] to handle elementary functions using cosine, sine, and exponential functions. This theory contains theorems about preserving continuity under certain operations together with the base cases of constant functions and the identity function being continuous. The theory is parameterised with a type `T`, which is used as the domain type for the functions.

We add the following theorems in order to work with functions including some of the transcendental functions:

```
exp_fun_continuous_lemma2: LEMMA
```

```
  continuous(f, x) IMPLIES continuous(exp(f), x)

cos_fun_continuous_lemma2: LEMMA
  continuous(f, x) IMPLIES continuous(cos(f), x)

sin_fun_continuous_lemma2: LEMMA
  continuous(f, x) IMPLIES continuous(sin(f), x)
```

We are not actually using the theorem about composition preserving continuity directly (only if the user wishes to prove continuity of functions involving uninterpreted functions such as `continuous(f(exp(x)))` would that be needed), but for each of our operators $+$, $-$, $*$, $/$, $|\cdot|$, exp, cos and sin we prove that for any continuous function(s) f (and g) then well-formed application of the operator preserves continuity. The main reason for this implicit use of composition is pragmatic: when composing $f : T_1 \to T_2$ and $g : T_2 \to \mathbf{R}$ in PVS one needs not only to determine the domain of the composed function but also of g, that is T_2. Although this can be obtained by inspecting the composed functions it is a fairly complicated procedure, and except when using uninterpreted functions the implicit use of composition works well.

We wish to use the theorems in as automatic a way as possible, and so we have two options; either we write special strategies to direct the proof, or we provide enough information for PVS to match the theorems to the input. The next two sections shows two very different approaches to automation (based on strategies and judgements) and due to the regularity of our problem area, using judgements seems to be a particularly good approach.

6.3 Method 1: Using Strategies

In this section we describe our implementation of Method 1 which relies entirely on strategies to perform continuity checking. The top-level strategy which implements Method 1 is called `cts1`. The idea is to follow the high school method, in which we examine the function in a purely syntactic fashion and use the knowledge gained from that to determine which theorems to apply. For example, the function $f(x) = 3x + a$ requires 5 theorem applications:

Each of the functions $f_1(x) = x$, $f_2(x) = 3$, $f_3(x) = a$ are continuous -- this is proven by applying the theorems `id_lemma` and `const_lemma` (twice).

Also the function $f_4(x) = 3x$ is continuous by theorem `prod_lemma` and finally $f_5(x) = 3x + a$ is continuous by `sum_lemma`.

One disadvantage with this type of strategy is that it is specific to the job in hand, so that although checking for existence of a limit can be done in the same syntax-directed manner, another set of strategies would have to be written to do that. As discussed in Section 6.5 it is possible that one could write some kind of "meta-strategy" which would apply to a range of syntax-directed checks, thereby making this type of strategy more desirable.

6.3.1 Preprocessing

As explained in Section 6.2 we aim to make the use of this continuity checker as simple as possible. The strategy is called `cts1` for continuity, and is shown in Figure 6.2. We ensured that `cts1` takes no arguments as we deem this a necessary feature in a properly automated continuity checker.

The first step is to do some preprocessing to ensure that the current goal is of the correct form and to determine the domain type of the functions to be checked. This is quite a crude version of the continuity checker, and so works only on the three different types of goals explained in the introduction (and then of course, as continuity is in general undecidable, it may still fail to prove continuity).

The preprocessing then consists of two parts: First, we ensure that the goal is of one of the above forms. Then, in the first two cases, we determine the type representing the domain of the function and perform what we call *lambda-driving*; that is, we rewrite the description of the function f from say $\lambda x.x+3$ to $(\lambda x.x)+(\lambda x.3)$ pushing the λ's as deep as possible. Because PVS is strongly typed, we can determine the domain of the function by observation of the goal itself and the strategy language provides means to extract it. In the last case, we just do Skolemisation and simplification, then call the strategy on the resulting goal ($Q(x)$ in (6.3)). Once the preprocessing is done, another strategy `shuffle-forms`, which should only be used from within `cts1`, is called using the domain type of the function as a parameter.

```
(defstep cts1 ()
  (let ((fmla (formula (car (p-sforms (current-goal *ps*))))))
    (if (and (forall-expr? fmla)
             (eq (id (operator (expression fmla))) '|continuous|))
        (let ((act (format nil "~a" (print-type (type-value (car (actuals
                     (module-instance (resolution
                                       (operator (expression fmla))))))))))
          (then (skosimp)(rec-lambda-driving$ act)
                (shuffle-forms$ act)(rec-cts$ act)))
      (if (and (application? fmla)
               (eq (id (operator fmla)) '|continuous|))
          (let ((act (format nil "~a" (print-type (type-value (car (actuals
                       (module-instance (resolution (operator fmla))))))))))
            (then (skosimp)(rec-lambda-driving$ act)
                  (shuffle-forms$ act)(rec-cts$ act)))
        (if (and (forall-expr? fmla)
                 (eq (id (operator (expression fmla))) 'IMPLIES))
            (THEN (skosimp)(cts1$))
          (skip-msg "cts1 only useful on top level op of continuous"))))))
"Uses rec-lambda-driving and rec-cts to explicitly apply the high
 school method for continuity. Requires the first consequent to
 contain an application of 'continuous'."
"Explicitly applying the high school method for continuity.")
```

Figure 6.2: The strategy `cts1`

```
(defstep shuffle-forms ()
  (if (not (eq (p-sforms (current-goal *ps*)) NIL))
  (let ((fmla (formula (car (p-sforms (current-goal *ps*))))))
    (if (and (application? fmla)
             (eq (id (operator fmla)) '|continuous|))
        (skip)
        (THEN (HIDE 1)(shuffle-forms$))))
 (fail))
"Repeatedly hides consequent no. 1 until the first consequent
 contains an application of 'continuous' or until there are no
 more consequents."
"")
```

Figure 6.3: The strategy `shuffle-forms`

6.3.2 Locating the Right Consequent

The preprocessing in `cts1` might introduce new consequents above the one about continuity, and since our next strategy requires this to be the top consequent, we now wish to hide any new consequentss. The strategy `shuffle-forms` in Figure 6.3 does just that. It works by recursively hiding the first consequent until the right consequent is at the top. In general

this works well, however in some cases we might be hiding information which would otherwise be useful. Take for example the following theorem $\forall x \,.\, x \neq 0 \Rightarrow \texttt{continuous}(f, x)$. By Skolemisation and simplification this becomes $x = 0 \vee \texttt{continuous}(f, x)$, where $x = 0$ is the first element in the list of consequents. If we hide this particular consequent, we can no longer make use of the fact that x is non-zero, thus potentially making the proof of the continuity harder or even impossible. In most of our test cases (Appendix B) though the loss of information does not hinder the future proof, although it is easy to find examples such as the one above where hiding certain consequents would make the goal unprovable in PVS.

6.3.3 Applying Theorems from `continuous_functions`

The strategy `rec-cts` in Figure 6.4 is one large let-statement, and the body is essentially just a case-statement. The let-statement is used to instantiate each of the theorems in Dutertre's theory `continuous_functions` described in Section 4.3.3. The functions this strategy can handle are those constructed using identity and constant functions, and the following combinations: $-$ (unary and binary), $+$, $*$, $/$ and $|\cdot|$. The internal representation of such functions differs depending on what the top-level operation is: unary minus is a *unary application*, identity is a *name expression*, constant functions (when represented using the `K_conversion`) are *implicit conversions* and the rest are all applications. The case-statement is then used to distinguish firstly between the different representations and then between each of the function combinators. In each case the instantiated theorem from the outermost let-statement is applied, then `rec-cts` is applied to each part of the function. At times this will generate TCCs, in which case PVS tries to discharge them automatically. If they are not all automatically discharged, we try `grind` on the remaining TCCs. We do not perform any analysis on the sub-goals, and it seems that the functions on which this strategy fails are ones where a more careful proof would be required in order to discharge these TCCs.

The strategy described here is very basic in that it does not perform many checks ensuring that, for example, applications of theorems work. Thus the strategy is quite fragile both with respect to the user and any future changes to PVS. However, as we shall see in Section 6.4 the use of judgements means we do not need a strategy as elaborate as this, although it is still useful to have a strategy that does all the preprocessing for the user.

```
(defstep rec-cts (fctdomain)
(let ((domain (format nil "~a" fctdomain))
      (sum_1 (format nil "sum_continuous[~a]" fctdomain))
          ...
      (fmla (formula (car (p-sforms (current-goal *ps*)))))
      (err (format nil "rec-cts apply to +, -, *, /, abs, constant and Id")))
  (if (eq (id (operator fmla)) '|continuous|)
      (let ((expr (car (exprs (argument fmla)))))
        (if (unary-application? expr)(let ((sym (id (operator expr))))
              (if (eq sym '-)(let ((arg (format nil "~a" (argument expr))))
                    (BRANCH (USE neg_1 ("f" arg))((inner$ domain)(GRIND))))
                (skip-msg err)))
          (if (or (infix-application? expr)
                  (or (application? expr)(implicit-conversion? expr)))
              (let ((sym (id (operator expr))))
                (if (eq sym '+)
                    (let ((arg1 (format nil "~a" (car (exprs (argument expr)))))
                          (arg2 (format nil "~a"
                                        (car (cdr (exprs (argument expr)))))))
                      (BRANCH (USE sum_1 ("f1" arg1 "f2" arg2))
                              ((inner$ domain)(GRIND))))
                  (if (eq sym '-)(BRANCH (USE diff_1)((inner$ domain)(GRIND)))
                    (if (eq sym '*)(BRANCH (USE prod_1)((inner$ domain)(GRIND)))
                      (if (eq sym '/)(BRANCH (USE div_1)
                            ((BRANCH (SPLIT)((assert)
                                             (THEN (assert)
                                                   (rec-lambda-driving$ domain)
                                                   (rec-cts$ domain))
                                             (THEN (assert)
                                                   (rec-lambda-driving$ domain)
                                                   (rec-cts$ domain))
                                             (GRIND)))(GRIND)))
                        (if (eq sym '|abs|)
                            (BRANCH (USE abs_1)((inner$ domain)(GRIND)))
                          (if (eq sym '|K_conversion|)
                              (THEN (USE const_1)(GRIND$ :DEFS NIL)(FAIL))
                              (skip-msg err))))))))
            (if (name-expr? expr)
                (let ((sym (id expr)))
                  (if (eq sym 'I)(THEN (USE id_1)(GRIND$ :DEFS NIL)(FAIL))
                    (skip-msg err))) (skip-msg err)))))
          (skip-msg "rec-cts useful on top level op of continuous")))
"Recursively explicitly applies the high school method to prove
 continuity, provided the first consequent contains an application
 of 'continuous'.
 Takes as argument the function domain."
"")
```

Figure 6.4: The strategy `rec-cts`

6.3.4 Example

We will now illustrate how the strategy is applied to a goal and the output generated.

Example 6.3 *For $x \in \mathbf{R}_+$ consider the function $f(x) = x + \frac{2}{x}$ as in Example 6.1. We wish to prove that f is continuous at 3. In PVS that is expressed by the following lemma:*

```
example: LEMMA
  continuous[posreal](LAMBDA (x: posreal): x + 2 / x, 3)
```

Here $+$ is the top symbol, and so we first apply `sum_continuous`. *This then gives us two sub-goals:*

```
continuous[posreal](LAMDBA (x: posreal): x, 3)
continuous[posreal](LAMBDA (x: posreal): 2 / x, 3)
```

A Type-Correctness Condition (TCC) is also generated

```
FORALL (x: posreal): x /= 0
```

However, it is discharged automatically by PVS. Figure 6.5 shows a PVS session proving `continuous[posreal](LAMBDA (x: posreal): x + 2 / x, 3)`. *In order to show the steps of the process we call the* glassbox *version of* `cts1`, *that is* `cts1$`, *as explained in Section 4.5. However, we have left out many of the rewrites used, just keeping the main proof steps which illustrate the high school method.* □

6.3.5 Results

The strategy `cts1` does not support transcendental functions, but it does work well for simple rational functions. Below are a few example of the functions proven correct by `cts1`. They are all of the type *domain*, that is we are checking that the function is continuous on

```
PVS(24):
example :

  |-------
{1}   continuous[posreal](LAMBDA (x: posreal): x + 2 / x, 3)

Rule? (cts1$)
...
Using lemma sum_continuous[posreal],

{-1}  continuous(LAMBDA (x_147: posreal): x_147, 3) AND
       continuous(LAMBDA (x_148: posreal): 2 / x_148, 3)
       IMPLIES
       continuous((LAMBDA (x_147: posreal): x_147) +
                  (LAMBDA (x_148: posreal): 2 / x_148),
                 3)
  |-------
[1]   continuous[posreal]
          (((LAMBDA (x_147: posreal): x_147) +
             (LAMBDA (x_148: posreal): 2 / x_148)),
           3)
...
Simplifying, rewriting, and recording with decision procedures,
Splitting conjunctions,
Using lemma identity_continuous[posreal],
Skolemizing (with typepred on new Skolem constants),
Applying disjunctive simplification to flatten sequent,
...
Using lemma div_continuous2[posreal],
...
Using lemma const_continuous[posreal],
...
Using lemma identity_continuous[posreal],
...
Skolemizing (with typepred on new Skolem constants),
Applying disjunctive simplification to flatten sequent,
...
This completes the proof of example.1.
example.2 (TCC):
  |-------
{1}   FORALL (x1: real): x1 >= 0 IMPLIES x1 >= 0
[2]   continuous[posreal](LAMBDA (x: posreal): x + 2 / x, 3)
...
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of example.2.

Q.E.D.
```

Figure 6.5: Proving `continuous(LAMBDA x: x + 2 / x, 3)`

the whole of its domain.

$$\lambda x \,.\, x + \frac{2}{x} \;;\; x \in \mathbf{R} \setminus \{0\}$$

$$\lambda x \,.\, \frac{1}{x - |x|} \;;\; x \in \mathbf{R}_-$$

$$\lambda x \,.\, 5\left(\frac{1}{|x|+1} + \frac{a}{|x|+2}\right) - x \;;\; x,\, a \in \mathbf{R}$$

$$\lambda x \,.\, \frac{1}{x+1} \;;\; x \in \mathbf{R} \setminus \{-1\}$$

$$\lambda x \,.\, \frac{1}{x} \;;\; x \in (1,3)$$

However, for functions sufficiently complicated this strategy is not good enough. One such example is `example39` (Appendix B.1):

$$\lambda x \,.\, 5\left(\frac{1}{|x|+1} + \frac{2}{x - |x|}\right) - x^2 \;;\; x \in \mathbf{R}_- \tag{6.6}$$

This is because the generated sub-goals can not be solved by a simple application of `grind`, for example

```
FORALL (y: negreal): continuous(LAMBDA (x: negreal):
  5 * (2 / (x - abs(x))), y)
```

So the reason `cts1` does not handle more complicated functions well is that it is itself too simple to discharge all the non-trivial side conditions (in the form of sub-goals) arising. It would be possible to extend `cts1` to handle both more of the non-trivial TCCs and elementary functions, but we chose instead to provide Method 2 which uses judgements to solve exactly this problem of the non-trivial TCCs for a large class of functions.

6.4 Method 2: Using Type Judgements

In Section 6.3.5 we saw that although the strategy `cts1` could solve many simple problems, it failed on too complicated ones, largely due to the generated TCCs being themselves more complicated. Interaction with the PVS team at SRI International pointed us in the direction of using judgements (Section 4.4) to support PVS in discharging the TCCs. However, as we shall see in Section 6.4.4, this works so well that a strategy controlling the syntax-directed

proof is not necessary, as an application of `grind` with the appropriate arguments often then suffices to check continuity. Furthermore, this method solves more complicated problems than `cts1` does, and it is easier to adapt to work with transcendental functions and for other analytic properties too.

By giving judgements, for example

```
negreal_minus_nnreal_is_negreal:
  JUDGEMENT -(nx: negreal, nny: nnreal) HAS_TYPE negreal
```

we state that a certain kind of expression has a certain type, in this case that a negative real minus a non-negative real gives a negative real as a result. As described in Section 4.4 we then use the full power of PVS to prove the judgements correct.

For PVS to apply the theorems of `continuous_functions` automatically (for instance during an application of `GRIND`), the typechecker must be able to decide that the arguments are of the appropriate type, for example to apply `div_continuous`, it must know the divisor to be a non-zero function. If the right judgements have been proved, this will happen.

6.4.1 Judgements on Functions

The above judgement is concerned with the type of subtracting two real numbers, but what we need in order to use the theorems of `continuous_functions` is judgements about the the usual function combinations as higher order operators, for example subtraction of two functions, so that

$$f(x) = \frac{1}{-(|x|+1)-3} \tag{6.7}$$

will be matched to `div_continuous`.

When matching the function in (6.7) to `div_continuous` we get the following TCC

$$\forall x \in \mathbf{R} \,.\, \lambda x \,.\, -(|x|+1)-3 \neq \lambda x \,.\, 0$$

This TCC is not discharged automatically by PVS, however it can be proven using `grind`.

Dutertre [Dut96] already defined the higher order versions of $+$, $-$ (unary and binary), $*$, $/$, and $|\cdot|$, but we have added higher order versions of cosine, sine, and exponential functions:

```
exp(f): [ T -> real ] = LAMBDA x: exp(f(x))
cos(f): [ T -> real ] = LAMBDA x: cos(f(x))
sin(f): [ T -> real ] = LAMBDA x: sin(f(x))
```

We see here that we are using overloading of the names `exp`, `cos` and `sin`. As long as PVS can determine from the context which version of a name should be used, overloading is allowed and even widely used.

For the real numbers in PVS we have subtypes such as `negreal` ($\mathbf{R}_-$), nnreal ($\mathbf{R} \setminus \mathbf{R}_-$), nzreal ($\mathbf{R} \setminus \{0\}$), npreal ($\mathbf{R} \setminus \mathbf{R}_+$) and posreal ($\mathbf{R}_+$). We then define corresponding function types on $T \subseteq \mathbf{R}$:

```
posfun: TYPE = [ T -> posreal ]
negfun: TYPE = [ T -> negreal ]
npfun:  TYPE = [ T -> npreal ]
nnfun:  TYPE = [ T -> nnreal ]
nzfun:  TYPE = [ T -> nzreal ]
```

Now we can use these types to describe in the form of judgements how various operators affect the types of functions:

```
pg:        VAR posfun
ng:        VAR negfun
nng:       VAR nnfun
nph, npg: VAR npfun

npfun_plus_npfun_is_npfun:   JUDGEMENT +(nph, npg) HAS_TYPE npfun
npfun_minus_nnfun_is_npfun:  JUDGEMENT -(nph, nng) HAS_TYPE npfun
npfun_times_npfun_is_nnfun:  JUDGEMENT *(nph, npg) HAS_TYPE nnfun
npfun_div_posfun_is_npfun:   JUDGEMENT /(nph, pg)  HAS_TYPE npfun
npfun_div_negfun_is_nnfun:   JUDGEMENT /(nph, ng)  HAS_TYPE nnfun
minus_npfun_is_nnfun:        JUDGEMENT -(nph)      HAS_TYPE nnfun
```

We have also added similar judgements for `nzfun`, `posfun`, `negfun`, `nnegfun`, and judgements for the $|\cdot|$ and identity functions. PVS contains some judgements like these about

the real numbers, and we have copied those, lifting them to the function types. However, we also added many other judgements about real numbers as we found the need for them. We have also included some judgements about transcendental functions:

```
%function type for functions with range in -1 to 1
mod_le1_fun: TYPE = [ real -> mod_le(1) ]

cos_is_mod_le1_fun: JUDGEMENT cos HAS_TYPE mod_le1_fun
sin_is_mod_le1_fun: JUDGEMENT sin HAS_TYPE mod_le1_fun

exp_pos: JUDGEMENT exp(x: real) HAS_TYPE posreal
exp_fun_pos:
  JUDGEMENT exp(f: [ T -> real ]) HAS_TYPE real_fun_ops[T].posfun
```

Given our development of transcendental functions (Section 5) most of the judgements have quite simple proofs as they correspond closely to theorems already proven.

The way we determined which judgements to add is very pragmatic: running through our example files (Appendix B) we tried out our strategy `cts` on an example; if it failed we compared the functions in the example to the current set of judgements to decide which new judgements might be needed. Then we added those judgements, re-ran the proof of the example and iterated the process. Once we had identified sufficient judgements to allow `cts` to solve the example, we would then prove those judgements correct. Occasionally we might add a judgement which was already covered by existing judgements, but when type checking PVS detected this and gave a warning. This helped us to avoid adding superfluous judgements to the development, or wasting time proving them.

6.4.2 Top-level Strategy

Once we have the appropriate collection of judgements, we can prove that a function such as $\exp\left(\frac{1}{|x|+1}\right)$ is continuous by simply calling `grind` with the appropriate arguments. For $\exp\left(\frac{1}{|x|+1}\right)$ this would be:

```
(GRIND :DEFS NIL
```

```
(defstep cts ()
  (THEN
   (GRIND :DEFS NIL)(shuffle-forms)
   (let ((fmla (formula (car (p-sforms (current-goal *ps*))))))
     (if (and (application? fmla)
              (eq (id (operator fmla)) '|continuous|))
         (let ((act (format nil "~a"
                            (car (actuals (module-instance (resolution
                                                            (operator fmla))))))))
           (let ((lambda_abstractions (format nil "lambda_abstractions[~a]"
                                              act))
                 (continuous_functions (format nil "continuous_functions[~a]"
                                               act))
                 (transc_cont (format nil "transc_cont[~a]" act))
                 (continuous (format nil "continuous[~a]" act))
                 (continuity_def (format nil "continuity_def[~a]" act)))
             (GRIND$ :DEFS NIL
                     :THEORIES (lambda_abstractions continuous_functions
                                transc_cont "trig_types" "aux")
                     :EXCLUDE (continuous continuity_def))))
         (skip-msg "cts only useful on top level op of continuous"))))
"Uses grind and the high school method to prove the first consequent
 containing an application of 'continuous'. If no such consequent is
 found, it fails."
"Applying the high school method for continuity.")
```

Figure 6.6: The strategy `cts`

```
:THEORIES ("lambda_abstractions[real]"
           "limit_of_functions[real]" "convergent_transc[real]"
           "transc" "trig_types" "aux" "isolated_points")
:EXCLUDE  ("adh_ext[real]" "adh_T[real]" "convergent[real]"
           "exp" "cos" "sin" "log" "tan"))
```

However, as before we wish the checker to be as automatic as possible, so we write a strategy `cts` (Figure 6.6) which does all the preprocessing, so that the instantiations are done automatically. It works much like the top-level strategy `cts1`, except it just calls `grind` rather than a purpose-built strategy. Also, we do not need to distinguish between different forms of the theorem to be proven, as `grind` itself can handle this.

```
PVS(47):
example :

  |-------
{1}   continuous[posreal](LAMBDA (x: posreal): x + 2 / x, 3)

Rule? (cts)
lambda_id rewrites LAMBDA (x_113: posreal): x_113
  to I[posreal]
lambda_const_fun rewrites LAMBDA (x: posreal): 2
  to K_conversion[real, posreal](2)
lambda_div2 rewrites LAMBDA (x_114: posreal): 2 / x_114
  to (K_conversion[real, posreal](2) / I[posreal])
lambda_add rewrites LAMBDA (x: posreal): x + 2 / x
  to (I[posreal] + (K_conversion[real, posreal](2) / I[posreal]))
identity_continuous rewrites continuous(I[posreal], 3)
  to TRUE
const_K_continuous rewrites continuous(K_conversion[real, posreal](2), 3)
  to TRUE
div_continuous rewrites
  continuous((K_conversion[real, posreal](2) / I[posreal]), 3)
  to TRUE
sum_continuous rewrites
  continuous[posreal]
      ((I[posreal] + (K_conversion[real, posreal](2) / I[posreal])), 3)
  to TRUE
...
Applying the high school method for continuity.,
Q.E.D.
```

Figure 6.7: Proving `continuous(LAMBDA x : x + 2 / x, 3)`

6.4.3 Example

The underlying technique is still the high school method.

Example 6.4 *We consider again the function* `continuous(LAMBDA x: x + 2 / x, 3)` *for* $x \in \mathbf{R}_+$ *from Example 6.3. The PVS call is as before, but this time we use the strategy* `cts` *rather than* `cts1`. *The result can be found in Figure 6.7. We see that when using* `cts` *with the judgements rather than* `cts1` *we automatically get just a trace of which theorems are applied – this corresponds to the steps in the high school method.* □

6.4.4 Results

As the strategy `cts` relies on judgements throughout the transcendental library to facilitate all the necessary matchings, it was fairly easy to support checking elementary functions by adding the appropriate judgements, and we have proven continuity of these functions over the stated domains using `cts`:

$$\lambda x\,.\ \exp(x^2 + |1 - x|) \qquad ; x \in \mathbf{R}$$

$$\lambda x\,.\ \exp(\cos(x) + 1) \qquad ; x \in \mathbf{R}$$

$$\lambda x\,.\ 5\left(\frac{1}{|x|+1} + \frac{a}{|x|+2}\right) - x \ ; x,\ a \in \mathbf{R}$$

$$\lambda x\,.\ \frac{1}{x - |x|} \qquad ; x \in \mathbf{R}_-$$

$$\lambda x\,.\ \frac{1}{x - 4} \qquad ; x \in \mathbf{R} \setminus \{4\}$$

$$\lambda x\,.\ \pi - 1 - \pi \exp(1 - \cos(x)) \quad \text{is continuous at } \arccos\left(1 - \ln\left(\frac{1+\pi}{\pi}\right)\right)$$

Furthermore it also proves (6.6) which `cts1` could not handle. So adding the judgements and using the simpler strategy `cts` solves more problems that the original strategy `cts1` alone.

Limitations of Method 2

However, as judgements are used at the typechecking level of PVS, the language used for expressing the judgements is quite restricted. For example, we might like to prove that

$$f(x) = \frac{1}{\cos(x) + 2} \tag{6.8}$$

is continuous at $x = 3$. We know from the judgement `cos_is_mod_le1_fun` that the type of $\cos(x)$ is `mod_le1`, that is $\cos(x)$ is in the closed interval $[-1, 1]$. Now we can define two new types

```
nmod1: NONEMPTY_TYPE = { (x: real) | x < -1 OR 1 < x}
nmod1_fun: NONEMPTY_TYPE = [ real -> nmod1 ]
```

the first of which contains those reals not in $[-1, 1]$, and the second contains the functions from $\mathbf{R}$ to `nmod1`. Unfortunately the current version of PVS does not allow the generalisation we would like here, so the typechecker can not automatically assert that 2 is of the type `nmod1`. Some facility to allow the typechecker to use the PVS evaluator to determine this kind of membership is likely to be added to a later version of PVS[1]. However, we can add another judgement stating that 2 is indeed in `nmod1`:

```
2nmod1: JUDGEMENT 2 HAS_TYPE nmod1
```

The following judgement states that $\cos(x) + g(x)$ is non-zero for g in `nmod1_fun`.

```
g: VAR nmod1_fun

sums_cosnsin_fun: JUDGEMENT +(cos, g) HAS_TYPE nzfun
```

and now PVS knows from the judgement `2nmod1` that 2 is in `nmod1`, and thus that the constant function 2 is in `nmod1_fun`, so `cts` can actually prove that the function in (6.8) is continuous at $x = 3$. Obviously, having to add a judgement for each constant is not an option, although this example shows that given the generalisation `cts` would work much better. For the time being, `cts` can not really handle examples such as this.

6.5 Other Analytic Properties

What we have called *the high school method* is a simple syntactic analysis of functions, combined with the well-known observation that continuity is preserved by the usual operations on functions. However, the same is true for several other analytic properties, such as convergence (existence of a limit), Lipschitz conditions (used for solving differential equations) and differentiability. These are all properties which can be considered using the high school method. In this section we will show how minor modifications to the strategies `cts` and `shuffle-forms` used for continuity checking in Section 6.4 enables us to check for convergence of functions too.

[1]Personal communication with PVS development team.

```
(defstep conv-check ()
  (THEN (GRIND :DEFS NIL)(shuffle-forms-limits)
        (let ((fmla (formula (car (p-sforms (current-goal *ps*))))))
          (if (and (application? fmla)
                   (eq(id (operator fmla)) '|convergent|))
              (let ((act (format nil "~a"
                                 (car (actuals (module-instance (resolution
                                                 (operator fmla))))))))
                (let ((lambda_abstractions (format nil "lambda_abstractions[~a]"
                                                   act))
                      (limit_of_functions (format nil "limit_of_functions[~a]"
                                                  act))
                      (convergent_transc (format nil "transc_conv[~a]" act))
                      (adh_ext (format nil "adherence_extend_T[~a]" act))
                      (adh_T (format nil "adherence_T[~a]" act))
                      (convergent (format nil "convergent[~a]" act)))
                  (GRIND$ :DEFS NIL
                          :THEORIES (lambda_abstractions limit_of_functions
                                     convergent_transc "transc" "trig_types"
                                     "aux" "isolated_points")
                          :EXCLUDE (adh_ext adh_T convergent "exp" "cos" "sin"
                                    "log" "tan"))))
              (skip-msg "cts only useful on top level op of convergent"))))
"Uses grind and the high school method to prove the first consequent
 containing an application of 'convergent'. If no such consequent is
 found, it fails."
"Applying the high school method for existence of limits.")
```

Figure 6.8: The strategy conv-check

So we consider the question: Does the function f have a limit at x, that is: is f convergent at x? Within Dutertre's analysis library, this can be stated as

```
example: LEMMA convergent(f, x)
```

Just as the basic theorems about preservation of continuity under the usual operations, Dutertre implemented theorems about preservation of convergence of functions. So we only need to modify `cts` very slightly for it to become `conv-check` (Figure 6.8). In order for `conv-check` to work we do not need any new theorems or judgements. The existing judgements are about function types rather than about continuity and so apply to `conv-check` too. Now we can prove, for example, that the function

$$f(x) = -\pi - 1 + \pi * \exp(1 - \cos(x)) \tag{6.9}$$

```
(defstep analysis ()
  (TRY-BRANCH (then (conv-check$)(fail))(skip)(cts$)))
"Uses conv-check to prove the first consequent containing an application
 of 'convergent'. If that fails, uses cts to prove the first consequent
 containing an application of 'continuous'. If that also fails, analysis
 fails."
"Applying the high school method for convergence/continuity.")
```

Figure 6.9: The strategy `conv-check`

has a limit at the point

$$x = \arccos\left(1 - \ln\left(\frac{1+\pi}{\pi}\right)\right) \tag{6.10}$$

6.5.1 Generalisation of `cts` and `conv-check`

As we have seen two nearly identical strategies solving two different problems, convergence and continuity, an obvious question is now: can we combine these two strategies into one? We did some simple tests to see if this might be a good idea. However, even with only two different options in the strategy, it is again more complicated and seems to be harder to extend – although not as bad as `cts1`. From the user's point of view there is perhaps not much point as one would have to type not only the strategy name but also the option, and the combined strategy seems slower than each of the specialised ones. These tests were fairly basic and it is very possible that one could develop a more sophisticated strategy, which handles various analytic checks, and is no less efficient than the specialised strategies while being easy to extend.

Another approach to combining the strategies, is to keep them as individual strategies and then combine those using the strategy language of PVS. This can be done very simply with the strategy in Figure 6.9. The strategy first tries to prove the goal by assuming it is about convergence of functions and using the strategy `conv-check`. However, even if the goal was not about convergence of a function, running `conv-check` is still likely to have changed the goal, which is why we use `(then ... (fail))` around the call to `conv-check` as this allows us to backtrack to the original goal. Then we try using the strategy `cts`. If none of the strategies apply, then `analysis` fails as usual.

When using this combined strategy a bit of time is lost in the case of continuity, as we first attempt to check for convergence. However, it does make the automation a bit easier to

use, although we would still have to write additional strategies to extend this to cover other analytic properties.

6.6 Discussion

In this section we discuss some of the limitations of the techniques we have developed, in particular we describe how not being able to *calculate* values is a hindrance. However, this is not unique to PVS but is a general problem in theorem proving.

6.6.1 Calculating Values

Theorem provers are in general not well suited for calculations. In our work this shows in that whereas we can prove that a function has a limit at a certain point, actually finding that limit, or even proving what it is is not easy. We made some experiments with differentiation to try and calculate the derivative, and our conclusion based on these experiments is that the lack of support for calculation prevents us from pursuing this further.

First we defined a new datatype of functions:

```
function_datatype[ T: TYPE FROM real ]: DATATYPE
  BEGIN
    const(x: real): const?
    id: id?
    add(g1: function_datatype, g2: function_datatype): add?
    mul(g1: function_datatype, g2: function_datatype): mul?
    sub(g1: function_datatype, g2: function_datatype): sub?
  END function_datatype
```

and then we defined the differentiation function on this datatype:

```
diff(f: function_datatype): recursive [ T -> real ] =
  cases f of
    const(x): lambda (y: T): 0,
    id: lambda (y: T): 1,
    add(g1,g2): lambda (y: T): diff(g1)(y) + diff(g2)(y),
    mul(g1,g2): lambda (y: T):
```

```
      diff(g1)(y) * eval(g2)(y) + diff(g2)(y) * eval(g1)(y),
    sub(g1,g2): lambda (y: T): diff(g1)(y) - diff(g2)(y)
  endcases
 MEASURE size(f)
```

where `size` is a function which gives a size to each element of the datatype, and `eval` is a function which gives the equivalent real-valued function.

These definitions then allows us to prove for instance that the derivative of $f(x) = x^2 + 3x$ is $2x + 3$:

```
test1: LEMMA % expressing f in the datatype
  eval(add(mul(id, id), mul(id, const(3)))) = lambda (y: T):  y * y + 3 * y

test2: LEMMA % calculating the derivative
  diff(add(mul(id, id), mul(id, const(3)))) = lambda (y: T):  2 * y + 3
```

We can prove a general correctness theorem about our new definition:

```
correctness_theorem: THEOREM
  forall (f: function_datatype):  deriv(eval(f)) = diff(f)
```

where `deriv` is Dutertre's definition giving the value of the derivative.

Whereas we can convert easily from our new function datatype to our usual real-valued function type `[ T -> real ]` going the other way is not possible, as the usual function types do not have obvious structure as the new one. The only way to try and determine the structure of an ordinary real-valued function is by using strategies to act as a parser from the ordinary real-valued functions to our new function type. We did not pursue this further. A major problem at this point is that to introduce division into our new function type we get the usual problem of what to do with division by zero. If we could somehow inherit information from the ordinary real-valued functions this would not be too disadvantageous, but as we have just seen this is not in general possible. So the approach using a new function datatype does not seem worthwhile as it will not work with existing theories.

So let us instead turn to simply proving that the derivative is what we expect it to be, for example:

```
test3: LEMMA deriv(lambda (x: T): x * x + 3 * x, 1) = 5
```

that is *the derivative of* $f(x) = x^2 + 3x$ *at 1 is 5*. Using existing lemmas by Dutertre we can prove this interactively by providing the right instantiations of the three lemmas needed. However, `grind` determines the correct lemmas to use, but gets the instantiations wrong. This is hardly surprising as it would have to guess how to split the number 5, so that sub-proofs become `deriv(lambda (x: T): x * x, 1) = 2` and `deriv(lambda (x: T): 3 * x, 1) = 3`. This also indicates that using an external system (for example a CAS) to calculate the derivative and then verify it in PVS might not be feasible either, unless the external system also provides information as to how the result was obtained.

Not being able to calculate values as limits and derivatives is an actual limitation on the complexity of problems we can attempt to solve automatically. For instance, anything involving L'Hôpital's Rule (Theorem 6.1) is nearly impossible as it involves calculating derivatives.

6.6.2 Functions Defined by Cases

Neither of our implementations of Method 1 or Method 2 can handle functions defined by cases. There are two main reasons why we decided to make this restriction. Firstly, deciding which case to use could be very difficult - without any restrictions it would be undecidable - and secondly, calculating the limit(s) at the splits is very hard as we saw in Section 6.6.1, so determining continuity at splits is not currently possible.

Consider the function

$$f(x) = \begin{cases} g(x) & \text{if B(x),} \\ h(x) & \text{otherwise.} \end{cases} \tag{6.11}$$

where g and h are functions not defined by cases and B is a predicate. Then B could be arbitrarily complicated, indeed it could be The Halting Problem. But even if we restrict B to say a linear inequality, such as $x \geq a$, we will still not be able to determine whether f is continuous at a, whereas we could possibly prove continuity for all other points. In order to do that with either of Method 1 or Method 2 with the current implementations, the question would have to be re-phrased from

$$\forall x \neq a \,.\, f \text{ is continuous at } x$$

to

$$(\forall x < a \,.\, h \text{ is continuous at } x) \wedge (\forall x > a \,.\, g \text{ is continuous at } x)$$

Then if we consider each of these as a separate theorem, they can be attempted with either of Method 1 or Method 2.

6.6.3 Removable Singularities

Part of our motivation for doing continuity checking is to support CASs in definite integration and solving differential equations (see Section 2.1.2). In both these cases, we do not necessarily require continuity everywhere but allow removable singularities.

A function f has a singularity [Apo74] at a point a in its domain if it is not differentiable at that point, but is differentiable on an open neighbourhood of a not containing a. The singularity is removable if there is a value which if assigned to $f(a)$ makes f differentiable at a. For example, the absolute value function has a singularity at 0, but it is not removable, as no change of value at 0 would make the function differentiable. The function

$$f(x) = \begin{cases} x^2 & \text{if } x \neq 3, \\ 0 & \text{otherwise.} \end{cases} \tag{6.12}$$

has a singularity at $x = 3$, but if we assign $3^2 = 9$ to $f(3)$ the function is differentiable at 3 and so the singularity of the original function is removable.

It is well-known that for example definite integration can ignore removable singularities – they do not change the value of a definite integral. So it would be interesting if we could use PVS to prove that a singularity is indeed removable. However, to do so we would again need to reason about actual values of functions, and as we have seen this is not really feasible.

6.7 Summary

The automatic continuity checker deals with pointwise continuity for a wide range of combinations of functions, including some from the library of transcendental functions. We

have seen two fundamentally different techniques implementing the high school method, which is syntax-directed; one which requires construction of highly specialised strategies, and one which relies on judgements to facilitate matching. Of these two methods, the latter is by far the most successful, easily proving theorems the first one failed to prove. Furthermore, the judgements are mostly quite simple and rarely more than one line, so they are easier to understand than a complicated strategy. Given the extended real analysis library, the judgements are also quite easy to prove, as they are either very simple or correspond to already proven theorems which can then be used in the proof of the judgement. It was also surprisingly easy to add support for transcendental functions by providing the few needed judgements, whereas with the purpose-built strategy these extensions would have been much harder. The obvious limitation of the judgements is with functions such as (6.8), since it is not currently possible to state usable general judgements which state that the divisor is non-zero. As we can prove theorems stating these properties within PVS, it is possible that a purpose-built strategy would do better on these more complicated cases, although we believe it is better to wait for PVS3 which is likely to provide the support needed to generalise the judgements as necessary for such cases[2].

We have also seen how fairly simple changes to the strategy `cts` allows us to use the judgements to check convergence of functions using the new strategy `conv-check`. Although the two strategies are almost identical, it does not seem feasible to construct a direct abstraction of them. However PVS's strategy language allows use of the command `try`, which allows us to construct a simple one-command strategy which combines `cts` and `conv-check` by first applying one, then if that fails applying the other. In this case of course both strategies have to exist, but then they are easy to combine.

As far as we are aware there are no other totally automated checkers for analytic properties. Given the basis of real analysis in a TP with a strategy language (for example HOL Light, Isabelle or Coq) this could be implemented. The nearest implementation we know of is Weierstrass [Bee98] by Beeson (see Section 2.3) which uses bounds to work out ε–δ proofs of continuity. However, Weierstrass is built inside Mathematica and uses Mathematica for calculations, and so is not meant to be trusted like a theorem prover is, rather it produces human-readable proofs which should then be checked by people. Beeson refers to this as Weierstrass producing proofs of *peer-review standard.*

[2]Personal communication with PVS development team.

Chapter 7

Conclusions and Further Work

In this thesis we have presented a PVS library supporting transcendental functions and a set of strategies for completely automatic checking of continuity and existence of limits of a large class of real-valued functions. There are some natural extensions to this work, both relating to the class of functions and the range of analytic properties covered by the automation. In this chapter we will first discuss our achievements and then suggest some directions for further work.

7.1 Assessment

We have implemented an extensive library of real analysis with transcendental functions and proven a large collection of facts about these functions. As we have seen with the AILS project mentioned in Section 2.3.3 doing continuous mathematics rather than discrete can greatly improve results in verification, and our library can be used in applications such as AILS. Thus the library can be used in verification when using PVS as an interactive theorem prover.

However, we have also implemented strategies to check some analytic properties, namely continuity and limits, of a restricted yet useful set of functions. This means that for those two areas of mathematical analysis within the restricted set of functions, PVS now supports totally automatic proofs, thus it is well-suited to be used with other software, such as a computer algebra system. Using the Maple-PVS interface (see Section 2.4.3) we have

experimented with using our library to check analytic side conditions on solutions to differential equations.

Our method of providing automation is based on adding type judgements, which are then proven, to PVS theories. As type judgements can express for instance that addition restricted to positive valued functions returns positive valued functions, the same type judgement applies in various different situations, such as when considering limits and also when considering continuity. Thus, the use of type judgements is a very general mechanism. As the type judgements are applied by the typechecker, the language used to express type judgements is restricted. However, as we have seen, it is still possible to cover many functions using type judgements like the one above about addition of positive functions.

Our work with real analysis in PVS has led us to identify the following four challenges.

1. The one major problem we found with the type judgements is that of general judgements involving numerals, that is whereas PVS "knows" that 2 is non-zero and thus is in `nzreal`, we can not automatically get similar "knowledge" about other user-defined types. For example, we define the type `nmod1` to be all reals which are either strictly less than -1 or strictly greater than 1. Whereas PVS can easily assess that say $1 < 2$, this does not happen at the typechecking level. So although a simple evaluation shows that 2 is in `nmod1`, we can not currently use this in PVS. Facilities to alleviate this exact problem are likely to be included in the next version of PVS.

2. The library, particularly at the lower levels, is not well organised. This is partly due to our initial inexperience with PVS and partly due to the fact that many of the theories are parameterised, and with several layers of parameterised theories it can become difficult for PVS to detect automatically which instantiation we intended to use. So in some cases, it is easier to include all the theories needed directly, but this then leads to a library structure which is not as transparent as is desirable.

3. Another organisational issue is the size of some of the theories. As PVS works with a proof file for each theory file, the proof files may become very large for large theories, and it is more feasible to keep the theories smaller if the proofs are long.

4. As so often is the case with first attempts, we would have liked to have taken a thorough real analysis textbook and worked through the basic definitions and theorems, comparing them to Dutertre's library. We do not suggest that the library should be

disregarded, since it forms a good basis of real analysis in PVS, but as we found the definition of limits of functions to be incorrect, it might be beneficial to go through at least the definitions again. However, we have no particular reason to believe that there are any more fundamental errors in Dutertre's library.

7.2 Further Work

In Chapter 6 we explained how we use what we have called the *high school method* to check for both continuity and existence of limits for functions. It works by simply observing that the property holds for some base functions (in either of our cases, these are constant and identity functions) and that when using certain well-defined combinations of functions with the property, the property is preserved. It is no surprise then that the high school method also applies to other analytic properties, apart from continuity and limits. Of particular interest are differentiability and Lipschitz conditions, which may be used as a criteria on input functions to differential equations ensuring the existence of a solution. It would be useful to add these and possibly other properties to the automation. There is currently no support for Lipschitz conditions in PVS, so that would have to be defined and all the basic theorems used in the high school method would have to be proven.

In Section 6.5.1 we discussed a first attempt at a general strategy, which runs the high school method either on continuity or existence of a limit, depending on the proof goal. However, with even more analytic properties this becomes even more desirable. We envisage a strategy which takes as argument the name of the property and then attempts to use the high school method with respect to that property on the proof goal. It is possible to express this in the PVS strategy language, although it is not immediately clear if the overhead would cause such a generalised strategy to be significantly less efficient than the current specialised strategies.

An obvious extension of the high school method is to include functions defined by cases, however to reason about say continuity at the points where the function changes from one case to another, we need to be able to do proofs with actual values. As explained in Section 6.6.1 this is fairly complicated and one would need to retain control of the actual proof, so using all-out strategies such as `grind` is probably not feasible. However, using the standard high school method and restriction of functions, we could still reason fairly

easily about functions defined by cases away from the splitting points.

Within PVS it is now clear that much can be achieved by using the type judgement facility carefully, however it is not clear what the limitations of this method are. Work is currently under way at SRI International on extending not so much the language for the type judgements, but the conclusions the typechecker can make based on the type judgements. This will make the type judgements even more powerful and thus probably with very little effort our automation could be extended to cover more functions.

Work is ongoing in our group at University of St Andrews using our library with Maple via the Maple-PVS interface developed within the group. As mentioned in Section 2.4.3 we are developing a safer continuity checker for Maple, but there are other interesting possible applications. For example, we could use PVS to verify other analytic properties, such as Lipschitz or differentiability, and then use that in applications, such as solving differential equations. This would require further work to be done both in Maple and in PVS, as the PVS library does not yet support Lipschitz conditions. This way we would be using PVS to handle correctly the analytical side conditions on the usual methods for differential equation solving that Maple does not handle.

One can imagine in the future automated theorem proving support for many more mathematical properties, such as various analytic properties, equation testing, and inequality testing. This work is only the start of making computational logic techniques available in automatic form in practical applications.

Bibliography

[ADG+01] A. A. Adams, M. Dunstan, H. Gottliebsen, T. Kelsey, U. Martin, and S. Owre. Computer algebra meets theorem proving: Integrating Maple and PVS. In Richard J. Boulton and Paul B. Jackson, editors, *14th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 27–42. Springer Verlag, September 2001. Online at `http://link.springer.de/link/service/series/0558/tocs/t2152.htm`.

[AGLM99] A. A. Adams, H. Gottliebsen, S. A. Linton, and U. Martin. VSDITLU: A verified symbolic definite integral table look-up. In H. Ganzinger, editor, *Automated Deduction—CADE-16*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 112–126. Springer-Verlag, 1999.

[Apo74] Tom M. Apostol. *Mathematical Analysis*. Addison-Wesley, 1974.

[BCZ98] Andrej Bauer, Edmund Clarke, and Xudong Zhao. Analytica — An experiment in combining theorem proving and symbolic computation. *Journal of Automated Reasoning*, 21(3):295–325, 1998.

[Bea97] Alan F. Beardon. *Limits*. Springer-Verlag, 1997.

[Bee89] Michael J. Beeson. Logic and computation in MathPert: An expert system for learning mathematics. In Erich Kaltofen and Stephen M. Watt, editors, *Computers and Mathematics*, pages 202–214. Springer-Verlag, 1989.

[Bee98] Michael J. Beeson. Automatic generation of epsilon-delta proofs of continuity. In Jacques Calmet and Jan Plaza, editors, *Artifical Intelligence and Symbolic Computation: International Conference AISC '98*, volume 1476 of *Lecture Notes in Artificial Intelligence*, pages 67–83. Springer-Verlag, 1998.

[Ber96] Laurent Bernardin. A review of symbolic solvers. *SIGSAM Bulletin (ACM Special Interest Group on Symbolic and Algebraic Manipulation)*, 30(1):9–20, March 1996.

[BHC95] Clemens Ballarin, Karsten Homann, and Jaques Calmet. Theorems and algorithms: An interface between Isabelle and Maple. In A. H. M. Levelt, editor, *ISSAC '95: Proceedings of the 1995 International Symposium on Symbolic and Algebraic Computation*, pages 150–157. ACM Press, 1995.

[BJK+97] B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuta, and D. Vasaru. A survey of the Theorema project. In Wolfgang W. Küchlin, editor, *ISSAC '97: Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation*, pages 384–391. ACM Press, 1997.

[Bra90] Robert L. Brabenec. *Introduction to Real Analysis*. PWS-KENT Publishing Company, 1990.

[Bro96] Manuel Bronstein. *Symbolic Integration 1*. Springer-Verlag, 1996.

[Bro98] Manuel Bronstein. Issac '98: Symbolic integration tutorial. Available at `http://www-sop.inria.fr/cafe/Manuel.Bronstein/bronstein-eng.html`, 1998.

[CG00] Alberto Ciaffaglione and Pietro Di Gianantonio. A co-inductive approach to real numbers. In T. Coquand, P. Dybjer, B. Nordström, and J. Smith, editors, *Types for Proofs and Programs*, volume 1956 of *Lecture Notes in Computer Science*, pages 114–130. Springer-Verlag, 2000.

[CGG+92] Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, Benton L. Leong, Michael B. Monagan, and Stephen M. Watt. *First Leaves: A Tutorial Introduction to Maple V*. Springer-Verlag, 1992.

[Che80] Gregory Cherlin. Rings of continuous functions: Decision problems. In L. Pacholski, J. Wierzejewski, and A. J. Wilkie, editors, *Model Theory of Algebra and Arithmetic: Proceedings of the Conference on Applications of Logic to Algebra and Arithmetic Held at Karpacz, Poland, September 1–7, 1979*, volume 834 of *Lecture Notes in Mathematics*, pages 44–91. Springer-Verlag, 1980.

[CM00] Víctor Carreño and César Muñoz. Aircraft trajectory modeling and alerting algorithm verification. In Harrison and Aagaard [HA00], pages 90–105.

[Dew00] Mike Dewar. OpenMath: An overview. *SIGSAM Bulletin (ACM Special Interest Group on Symbolic and Algebraic Manipulation)*, 34(2):2–5, June 2000.

[Dut96] Bruno Dutertre. Elements of mathematical analysis in PVS. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference*, volume 1125 of *Lecture Notes in Computer Science*, pages 141–156. Springer-Verlag, 1996.

[Dwi57] Herman Bristol Dwight. *Tables of Integrals and Other Mathematical Data.* Macmillan, 1957.

[Fle00] Jacques D. Fleuriot. On the mechanization of real analysis in Isabelle/ HOL. In Harrison and Aagaard [HA00], pages 146–162.

[GCL92] Keith O. Geddes, Stepehn R. Czapor, and George Labahn. *Algorithms for Computer Algebra.* Kluwer, 1992.

[GM93] M. J. C. Gordon and T. F. Melham. *Introduction to HOL.* Cambridge University Press, 1993.

[GMW79] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[GPWZ00] Herman Geuvers, Randy Pollack, Freek Wiedijk, and Jan Zwanenburg. Skeleton for the proof development leading to the fundamental theorem of algebra, 2000. Available at `http://www.cs.kun.nl/~herman/FTA.mathproof.ps.gz`.

[HA00] J. Harrison and M. Aagaard, editors. *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[Har98] John Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998.

[Har00] John Harrison. Formal verification of IA-64 division algorithms. In Harrison and Aagaard [HA00], pages 232–251.

[HT93] J. Harrison and L. Théry. Reasoning about the reals: the marriage of HOL and Maple. In A. Voronkov, editor, *Logic Programming and Automated Reasoning (LPAR '93)*, volume 698 of *Lecture Notes in Artificial Intelligence*, pages 351–359. Springer-Verlag, 1993.

[Jac95] P. B. Jackson. *Enhancing the NUPRL Proof Development System and Applying it to Computational Abstract Algebra.* PhD thesis, Cornell University, April 1995.

[JS92] Richard D. Jenks and Robert S. Sutor. *AXIOM : The scientific computation system.* Springer-Verlag, 1992.

[Kap57] Wilfred Kaplan. *Advanced Calculus.* Addison-Wesley, 1957.

[KKS98] Manfred Kerber, Michael Kohlhase, and Volker Sorge. Integrating computer algebra into proof planning. *Journal of Automated Reasoning*, 21(3):327–355, 1998.

[KMM00] Matt Kaufmann, Panagiotis Manolios, and J. Strother Moore. *Computer-Aided Reasoning: An Approach.* Kluwer, 2000.

[Mad91] Tage Gutmann Madsen. *Matematisk Analyse.* Matematisk Institut, Aarhus Universitet, 1991.

[May99] Michaela Mayero. The three gap theorem: Specification and proof in Coq. Technical Report 3848, INRIA Rocquencourt, December 1999.

[MCB+] C. Muñoz, V. Carreño, R. Butler, G. Dowek, A. Geser, and B. Di Vito. A pragmatic nonfoundational theory of trigonometry. Available at `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/trig.dmp.gz`.

[Min98] Paul S. Miner. *Hardware Verification using Coinductive Assertions.* PhD thesis, Indiana University, June 1998.

[OSRSC99a] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference.* Computer Science Laboratory, SRI International, September 1999.

[OSRSC99b] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, September 1999.

[Pau93] Lawrence C. Paulson. Introduction to Isabelle. Technical Report 280, Computer Laboratory, University of Cambridge, 1993.

[Rez83] A. Rezus. *Abstract Automath*. Mathematisch Centrum, 1983.

[Rud92] Piotr Rudnicki. An overview of the MIZAR project, 1992.

[Rus98] David M. Russinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. *LMS Journal of Computation and Mathematics*, 1:148–200, 1998.

[SORSC99] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, September 1999.

[Tea] The Coq Development Team. The Coq Proof Assistant, Reference Manual. Available at `http://pauillac.inria.fr/coq/doc/main.html`.

[TF88] George B. Thomas and Ross L. Finney. *Calculus and Analytic Geometry*. Addison-Wesley, 1988.

[Wol91] Stephen Wolfram. *Mathematica: a system for doing mathematics by computer*. Addison-Wesley, 1991.

Appendix A

Strategies

This appendix contains all our strategies used for checking continuity and convergence of functions. Appendix A.1 contains the strategy `cts1` which implements Method 1 (Section 6.3), Appendix A.2 the strategy `cts` implementing Method 2 (Section 6.4), Appendix A.3 the strategy `conv-check` which uses Method 2 to check for existence of limits and finally Appendix A.4 contains the strategy `analysis` which combines `cts` and `conv-check`.

A.1 The Strategy `cts1`

```
;; Strategy to do continuity checking via Method 1
;; Supposed to work without judgements

(defstep rec-lambda-driving (fctdomain)
  (let ((domain (format nil "~a" fctdomain))
        (sum_lemma (format nil "lambda_add[~a]" fctdomain))
        (diff_lemma (format nil "lambda_sub[~a]" fctdomain))
        (div_lemma (format nil "lambda_div2[~a]" fctdomain))
        (prod_lemma (format nil "lambda_mult[~a]" fctdomain))
        (abs_lemma (format nil "lambda_abs[~a]" fctdomain))
        (const_lemma (format nil "lambda_const_fun[~a]" fctdomain))
        (id_lemma (format nil "lambda_id[~a]" fctdomain))
        (inv_lemma (format nil "lambda_inv_fct[~a]" fctdomain))
        (neg_lemma (format nil "lambda_neg[~a]" fctdomain))
        (fmla (formula (car (p-sforms (current-goal *ps*))))))
    (if (eq (id (operator fmla)) '|continuous|)
        (let ((expr (car (exprs (argument fmla)))))
          (if (lambda-expr? expr)
              (let ((expr2 (expression expr)))
                (if (unary-application? expr2)
                    (let ((sym (id (operator expr2))))
                      (if (eq sym '-)
                          (BRANCH (USE neg_lemma)((THEN (assert)(replace -1 1)(hide -1))
                                                 (GRIND)))
                        (skip-msg "(1) rec-lambda-driving only used on rational fct")))
                  (if (or (infix-application? expr2)
                          (or (application? expr2)
                              (implicit-conversion? expr2)))
                      (let ((sym (id (operator expr2))))
                        (if (eq sym '+)
```

```
                (BRANCH (USE sum_lemma)((THEN (assert)(replace -1 1)(hide -1))
                                        (GRIND)))
              (if (eq sym '-)
                  (BRANCH (USE diff_lemma)((THEN (assert)(replace -1 1)(hide -1))
                                           (GRIND)))
                (if (eq sym '*)
                    (BRANCH (USE prod_lemma)((THEN (assert)(replace -1 1)
                                                   (hide -1))(GRIND)))
                  (if (eq sym '/)
                      (BRANCH (USE div_lemma)
                              ((BRANCH (SPLIT)
                                       ((THEN (assert)(replace -1 1)(hide -1))
                                        (GRIND)))
                               (GRIND)))
                    (if (eq sym '|abs|)
                        (BRANCH (USE abs_lemma)((THEN (ASSERT)(replace -1 1)
                                                      (hide -1))
                                                (GRIND)))
                      (skip-msg
                       "(2) rec-lambda-driving only used on rational fct"
                       )))))))
          (if (or (name-expr? expr2)
                  (number-expr? expr2))
              (if (eq (free-variables expr2) nil)
                  (THEN (THEN (USE const_lemma)(GRIND$ :DEFS NIL)))
                (if (eq (id (car (free-variables expr2))) (id expr2))
                    (THEN (THEN (USE id_lemma)(GRIND$ :DEFS NIL)))
                  (skip-msg "(3) rec-lambda-driving only used on rational fct")))
            (skip-msg "(3.5) rec-lambda-driving only used on rational fct")))))
      (skip-msg "(4) rec-lambda-driving only used on rational fct")))
  (skip-msg "(5) rec-lambda-driving only used on top level op of continous")))
"Recursively pushes lambdas in a function description in the first
 consequent inwards,
```

```
 e.g. applied to (lambda x : x+3) gives (lambda x : x + lambda x : 3).
 Takes as argument the function domain."
"")

(defstep inner (fctdomain)
  (let ((domain (format nil "~a" fctdomain)))
    (THEN (ASSERT)(BRANCH (SPLIT)((THEN (rec-lambda-driving$ domain)(rec-cts$ domain)))))
    )
"Uses branch to do lambda-driving (using rec-lambda-driving) followed
 by continuity checking (using rec-cts).
 Takes as argument the function domain"
"")

(defstep rec-cts (fctdomain)
  (let ((domain (format nil "~a" fctdomain))
        (sum_lemma (format nil "sum_continuous[~a]" fctdomain))
        (diff_lemma (format nil "diff_continuous[~a]" fctdomain))
        (div_lemma (format nil "div_continuous2[~a]" fctdomain))
        (prod_lemma (format nil "prod_continuous[~a]" fctdomain))
        (abs_lemma (format nil "abs_continuous[~a]" fctdomain))
        (const_lemma (format nil "const_continuous[~a]" fctdomain))
        (id_lemma (format nil "identity_continuous[~a]" fctdomain))
        (inv_lemma (format nil "inv_continuous[~a]" fctdomain))
        (neg_lemma (format nil "opp_continuous[~a]" fctdomain))
        (fmla (formula (car (p-sforms (current-goal *ps*))))))
    (if (eq (id (operator fmla)) '|continuous|)
        (let ((expr (car (exprs (argument fmla)))))
          (if (unary-application? expr)
              (let ((sym (id (operator expr))))
                (if (eq sym '-)
                    (let ((arg (format nil "~a" (argument expr))))
                      (BRANCH (USE neg_lemma ("f" arg))((inner$ domain)(GRIND))))
                  (skip-msg "rec-cts only useful for +, -, *, /, abs, constant and Id")))
```

```
(if (or (infix-application? expr)
        (or (application? expr)
            (implicit-conversion? expr)))
    (let ((sym (id (operator expr))))
      (if (eq sym '+)
          (let ((arg1 (format nil "~a" (car (exprs (argument expr)))))
                (arg2 (format nil "~a" (car (cdr (exprs (argument expr)))))))
            (BRANCH (USE sum_lemma ("f1" arg1 "f2" arg2))
                    ((inner$ domain)(GRIND))))
        (if (eq sym '-)
            (BRANCH (USE diff_lemma)((inner$ domain)(GRIND)))
          (if (eq sym '*)
              (BRANCH (USE prod_lemma)((inner$ domain)(GRIND)))
            (if (eq sym '/)
                (BRANCH (USE div_lemma)
                        ((BRANCH (SPLIT)
                                 ((assert)
                                  (THEN (assert)(rec-lambda-driving$ domain)
                                        (rec-cts$ domain))
                                  (THEN (assert)(rec-lambda-driving$ domain)
                                        (rec-cts$ domain))
                                  (GRIND)))
                         (GRIND)))
              (if (eq sym '|abs|)
                  (BRANCH (USE abs_lemma)((inner$ domain)(GRIND)))
                (if (eq sym '|K_conversion|)
                    (THEN (THEN (USE const_lemma)(GRIND$ :DEFS NIL))(FAIL))
                  (skip-msg "cts only used on rational fct")))))))))
  (if (name-expr? expr)
      (let ((sym (id expr)))
        (if (eq sym 'I)
            (THEN (THEN (USE id_lemma)(GRIND$ :DEFS NIL))(FAIL))
          (skip-msg "rec-cts only used on rational fct")))
```

```
              (skip-msg "rec-cts only used on rational fct")))))
            (skip-msg "rec-cts only used on rational fct")))
"Recursively explicitly applies the high school method to prove
 continuity, provided the first consequent contains an application
 of 'continuous'.
 Takes as argument the function domain."
"")

(defstep shuffle-forms ()
  (if (not (eq (p-sforms (current-goal *ps*)) NIL))
  (let ((fmla (formula (car (p-sforms (current-goal *ps*))))))
    (if (and (application? fmla)
     (eq (id (operator fmla)) '|continuous|))
(skip)
(THEN (HIDE 1)(shuffle-forms$))))
 (fail))
"Repeatedly hides consequent no. 1 until the first consequent
 contains an application of 'continuous' or until there are no
 more consequents."
"")

(defstep cts1 ()
  (let ((fmla (formula (car (p-sforms (current-goal *ps*))))))
    (if (and (forall-expr? fmla)
             (eq (id (operator (expression fmla))) '|continuous|))
        (let ((act (format nil "~a"
                           (print-type
                            (type-value
                             (car
                              (actuals
                               (module-instance
                                (resolution (operator (expression fmla))))))))))))
          (then (skosimp)(rec-lambda-driving$ act)
```

```
            (shuffle-forms$ act)(rec-cts$ act)))
      (if (and (application? fmla)
               (eq (id (operator fmla)) '|continuous|))
          (let ((act (format nil "~a"
                             (print-type
                              (type-value
                               (car
                                (actuals
                                 (module-instance (resolution (operator fmla))))))))))
            (then (skosimp)(rec-lambda-driving$ act)
                  (shuffle-forms$ act)(rec-cts$ act)))
        (if (and (forall-expr? fmla)
                 (eq (id (operator (expression fmla))) 'IMPLIES))
            (THEN (skosimp)(cts1$))
          (skip-msg "cts1 only used on top level op of continuous")))))
"Uses rec-lambda-driving and rec-cts to explicitly apply the high
 school method for continuity. Requires the first consequent to
 contain an application of 'continuous'."
"Explicitly applying the high school method for continuity.")
```

A.2 The Strategy cts

```
;; Strategy to do continuity checking via Method 2
;; Supposed to work with judgements

(defstep cts ()
  (THEN (GRIND :DEFS NIL)(shuffle-forms)
        (let ((fmla (formula (car (p-sforms (current-goal *ps*))))))
          (if (and (application? fmla)
                   (eq (id (operator fmla)) '|continuous|))
              (let ((act (format nil "~a"
                                 (car
                                  (actuals
                                   (module-instance (resolution (operator fmla))))))))
                (let ((lambda_abstractions (format nil "lambda_abstractions[~a]" act))
                      (continuous_functions (format nil "continuous_functions[~a]" act))
                      (transc_cont (format nil "transc_cont[~a]" act))
                      (continuous (format nil "continuous[~a]" act))
                      (continuity_def (format nil "continuity_def[~a]" act)))
                  (GRIND$ :DEFS NIL
                          :THEORIES (lambda_abstractions continuous_functions transc_cont
                                                         "trig_types" "aux")
                          :EXCLUDE (continuous continuity_def))))
              (skip-msg "cts only used on top level op of continuous"))))
"Uses grind and the high school method to prove the first consequent
 containing an application of 'continuous'. If no such consequent is
 found, it fails."
"Applying the high school method for continuity.")
```

A.3 The Strategy conv-check

```
;; Strategies to check for existence of limits (convergent(f,x))
;; Using Method 2
;; Supposed to work with judgements

(defstep shuffle-forms-limits ()
  (if (not (eq (p-sforms (current-goal *ps*)) NIL))
  (let ((fmla (formula (car (p-sforms (current-goal *ps*))))))
    (if (and (application? fmla)
             (eq (id (operator fmla)) '|convergent|))
        (skip)
        (THEN (HIDE 1)(shuffle-forms-limits$))))
  (fail))
"Repeatedly hides consequent no. 1 until the first consequent
 contains an application of 'convergent' or until there are no
 more consequents."
"")

;; For intervals, this will only work on intervals
;; on the form I2(a,b) where b > a

(defstep conv-check ()
  (THEN (GRIND :DEFS NIL)(shuffle-forms-limits)
        (if (not (eq (p-sforms (current-goal *ps*)) NIL))
            (let ((fmla (formula (car (p-sforms (current-goal *ps*))))))
              (if (and (application? fmla)
                       (eq (id (operator fmla)) '|convergent|))
                  (let ((act (format nil "~a"
                                     (car
                                      (actuals
                                       (module-instance (resolution (operator fmla))))))))
```

```
                (let ((lambda_abstractions (format nil "lambda_abstractions[~a]" act))
                      (limit_of_functions (format nil "limit_of_functions[~a]" act))
                      (convergent_transc (format nil "transc_conv[~a]" act))
                      (adh_ext (format nil "adherence_extend_T[~a]" act))
                      (adh_T (format nil "adherence_T[~a]" act))
                      (convergent (format nil "convergent[~a]" act)))
                  (GRIND$ :DEFS NIL
                          :THEORIES (lambda_abstractions limit_of_functions
                                                         convergent_transc
                                                         "transc" "trig_types"
                                                         "aux" "isolated_points")
                          :EXCLUDE (adh_ext adh_T convergent "exp" "cos" "sin"
                                            "log" "tan"))))
            (skip-msg "conv-check only useful on top level op of convergent")))
        (fail)))
"Uses grind and the high school method to prove the first consequent
 containing an application of 'convergent'. If no such consequent is
 found, it fails."
"Applying the high school method for existence of limits.")
```

A.4 The Strategy `analysis`

```
;; Strategy to check for existence of limit or continuity
;; Using Method 2
;; Supposed to work with judgements

(defstep analysis ()
  (TRY-BRANCH (then (conv-check$)(fail))(skip)(cts$))
"Uses conv-check to prove the first consequent containing an application
 of 'convergent'. If that fails, uses cts to prove the first consequent
 containing an application of 'continuous'. If that also fails, analysis
 fails."
"Applying the high school method for convergence/continuity.")
```

Appendix B

Example Files

This appendix contains our example files with a total of 88 examples and test results (see Section 6.1). For each lemma the tables show which of the three types (d for domain, p for point and c for precondition) the lemma has. We then tried each lemma with Method 1, Method 1 with judgements, Method 2 and the combined strategy `analysis` (for continuity), and Method 2 and the combined strategy (for convergence). For each test we show if it worked (with a time in seconds as explained below) or gave an error, alternatively we indicate that the method was not applicable for a lemma of that type.

All timings in this appendix refers to PVS's *run time* given for each completed proof. They are not true representatives of how well the various strategies do, as the time is affected too greatly by other processes on the computer. Timings are based on a single measurement, except in cases where that measurement seemed far off, for example 17 seconds rather than predicted 11 seconds. These were often due to a garbage collection taking place, and so we re-ran the proof. However, some proofs are so large they require (repeated) garbage collection, in which case re-running them did not significantly change the measured time.

Each section in this appendix contains one or more example files and the test results related to those examples. Appendix B.1 contains 55 lemmas, apart from four these are all about continuity of functions. Appendix B.2 contains two example files with lemmas relating to values and continuity checking involving cos or the type $\mathbf{R}\backslash\{4\}$. Appendix B.3 contains 17 lemmas of which all but two are about limits of functions. Finally, Appendix B.4 contains two example files with examples relating to differential equations.

B.1 Proving Continuity

```
continuity_examples: THEORY
 BEGIN

  IMPORTING top_analysis

  x, y, z, a: VAR real
  n0x, n0y: VAR nzreal
  px, py: VAR posreal
  nx, ny: VAR negreal
  f, g: VAR [real -> real]

  NOT_EQ(x): NONEMPTY_TYPE = {r: real | r /= x} CONTAINING x + 1

  x1, y1: VAR NOT_EQ(-1)

  example1: LEMMA
    continuous[posreal](LAMBDA (x: posreal): x + 2 / x, 3)

  example2: LEMMA continuous(LAMBDA x: x, y)

  example3: LEMMA continuous(LAMBDA x: a, y)

  example4: LEMMA continuous(LAMBDA x: -x, y)

  example5: LEMMA continuous(LAMBDA x: a + x, y)

  example6: LEMMA continuous(LAMBDA x: a - x, y)

  example7: LEMMA continuous(LAMBDA n0x: a + n0x + z, n0y)

  example8: LEMMA continuous(LAMBDA x: a * x, y)
```

```
example9: LEMMA continuous(LAMBDA x: a * x + 2 * x - 3 * x, y)

example10: LEMMA continuous(LAMBDA x: 3, y)

example11: LEMMA
  continuous(f, y) AND continuous(g, y) IMPLIES continuous(f + g, y)

example12: LEMMA continuous(LAMBDA n0x: 1 / n0x, n0y)

example13: LEMMA continuous(LAMBDA n0x: n0x + 1 / n0x, n0y)

example14: LEMMA continuous(LAMBDA n0x: n0x + 2 / n0x, n0y)

example15: LEMMA continuous(LAMBDA x: 3 * x + x * x, y)

example16: LEMMA continuous(LAMBDA x: abs(x), y)

example17: LEMMA continuous(LAMBDA px: 1 / px, py)

example18: LEMMA continuous(LAMBDA px: 1 / abs(px), py)

example19: LEMMA continuous(LAMBDA px: 1 / (abs(px) + px), py)

example20: LEMMA continuous(LAMBDA px: 1 / (px + 1), py)

example21: LEMMA continuous(LAMBDA px: 1 / (px + px), py)

example22: LEMMA continuous(LAMBDA x: 1 / (abs(x) + 1), y)

example23: LEMMA continuous(LAMBDA x: 1 / (abs(x) - abs(x) + 1), y)

example24: LEMMA continuous(LAMBDA x: 2 / (abs(x) + 2), y)

example25: LEMMA
```

```
  continuous(LAMBDA x: 1 / (abs(x) + 1) + 2 / (abs(x) + 2), y)

example26: LEMMA (LAMBDA nx: nx)(-2) = -2

example27: LEMMA continuous[negreal](LAMBDA nx: nx, -2)

example28: LEMMA continuous[negreal](LAMBDA nx: 2, -2)

example29: LEMMA
  continuous[negreal](LAMBDA nx: 1 / (abs(nx) + 1), ny)

example30: LEMMA
  continuous[negreal](LAMBDA nx: 1 / (1 + abs(nx)), ny)

example31: LEMMA
  continuous[negreal](LAMBDA nx: 1 / (nx - abs(nx)), ny)

example32: LEMMA continuous[negreal](LAMBDA nx: nx - abs(nx), ny)

example33: LEMMA continuous[negreal](LAMBDA nx: 1 / abs(nx), ny)

example34: LEMMA continuous[negreal](LAMBDA nx: 1 / (-abs(nx)), ny)

example35: LEMMA
  continuous[negreal](LAMBDA nx: 1 / ((-3) + (-abs(nx))), ny)

example36: LEMMA
  continuous[posreal](LAMBDA px: 1 / ((abs(px) + 1) - px), py)

example37: LEMMA continuous
    (LAMBDA x: 5 * (1 / (abs(x) + 1) + 2 / (abs(x) + 2)) - x * x, y)

example38: LEMMA continuous[negreal]
    (LAMBDA nx: 5 * (1 / (abs(nx) + 1) + 2 / (nx - abs(nx))), ny)
```

```
example39: LEMMA continuous[negreal](LAMBDA nx:
  5 * (1 / (abs(nx) + 1) + 2 / (nx - abs(nx))) - nx * nx, ny)

example40: LEMMA continuous
  (LAMBDA x: 5 * (1 / (abs(x) + 1) + 2 / (abs(x) + 2)) - x * x, y)

example41: LEMMA continuous[nzreal](LAMBDA n0x: 1 / n0x, 1)

example42: LEMMA
  FORALL (y: NOT_EQ(-1)):
    continuous(LAMBDA (x: NOT_EQ(-1)): 1 / (x + 1), y)

example43: LEMMA FORALL y1: continuous(LAMBDA x1: 1 / (x1 + 1), y1)

example44: LEMMA FORALL y1: continuous(I[NOT_EQ(-1)], y1)

I1: TYPE = {z: real | 1 < z AND z < 3}

example45: LEMMA
  FORALL (zi: I1): continuous(LAMBDA (xi: I1): 1 / xi, zi)

I2(b1: real, b2: real): TYPE = {z: real | b1 < z AND z < b2}

example46: LEMMA FORALL (a: real): a < a + 1

example47: LEMMA FORALL (p: real): p + 1 < p + 2 AND p + 2 < p + 3

example48: LEMMA continuous(LAMBDA x: x)

example49: LEMMA continuous(LAMBDA x: a)

example50: LEMMA continuous(LAMBDA x: -x)
```

```
  example51: LEMMA continuous(LAMBDA x: a + x)

  example52: LEMMA continuous(LAMBDA x: a - x)

  example53: LEMMA continuous(LAMBDA n0x: a + n0x + z)

  example54: LEMMA FORALL (a: real), (x: I2(a - 1, a + 1)): TRUE

  example55: LEMMA
    FORALL (x: {z: real | z /= 1}):
      continuous[{z: real | z /= 1}]
        (LAMBDA (y: {z: real | z /= 1}): 1 / (y - 1), x)

  example56: LEMMA continuous(LAMBDA px: 1/exp(px), py)

END continuity_examples
```

Name	Type	cts1	cts1+judgements	cts	conv-check	analysis
example1	p	8·98	9·36	11·58	n/a	11·08
example2	d	0·30	0·27	1·57	n/a	1·46
example3	d	0·34	0·32	1·35	n/a	1·50
example4	d	0·58	0·58	1·44	n/a	1·51
example5	d	0·80	0·79	1·54	n/a	1·64
example6	d	0·72	0·72	1·48	n/a	1·65
example7	d	1·54	1·43	1·64	n/a	1·75

Name	Type	cts1	cts1+judgements	cts	conv-check	analysis
example8	d	0·75	0·73	1·51	n/a	1·59
example9	d	2·91	2·96	1·58	n/a	1·73
example10	d	0·36	0·34	1·32	n/a	1·46
example11	c	0·31	0·29	1·41	n/a	1·59
example12	d	2·03	2·08	1·44	n/a	1·55
example13	d	2·92	2·92	1·63	n/a	1·73
example14	d	2·89	3·02	1·65	n/a	1·78
example15	d	1·82	1·83	1·77	n/a	1·83
example16	d	0·60	0·54	1·39	n/a	1·52
example17	d	4·24	4·37	11·24	n/a	11·06
example18	d	7·73	8·21	11·35	n/a	11·24
example19	d	13·11	14·19	12·03	n/a	12·00
example20	d	7·90	8·70	11·58	n/a	11·40
example21	d	8·14	8·96	11·55	n/a	11·77
example22	d	4·97	5·54	1·91	n/a	1·97
example23	d	2·17	2·16	1·56	n/a	1·68
example24	d	4·98	5·49	1·96	n/a	2·02
example25	d	14·61	15·76	2·52	n/a	2·56
example26		n/a	n/a	0·14	0·14	0·15
example27	p	0·61	0·58	9·78	n/a	9·92
example28	p	0·62	0·63	10·31	n/a	9·99
example29	d	12·53	14·07	12·13	n/a	11·95
example30	d	12·53	13·74	12·12	n/a	11·94
example31	d	13·53	14·68	12·00	n/a	12·03
example32	d	7·37	7·57	10·97	n/a	11·02
example33	d	7·70	8·49	11·62	n/a	11·68
example34	d	12·24	13·57	11·83	n/a	11·80
example35	d	17·95	19·50	12·49	n/a	12·48
example36	d	16·58	17·84	11·14	n/a	11·23
example37	d	32·66	34·44	3·09	n/a	2·98
example38	d	60·72	65·18	15·54	n/a	15·22
example39	d	error	278·17	16·57	n/a	16·09
example40	d	32·48	34·15	3·10	n/a	3·01

Name	Type	cts1	cts1+judgements	cts	conv-check	analysis
example41	p	1·99	2·10	1·33	n/a	1·35
example42	d	2·92	3·00	3·16	n/a	3·22
example43	d	2·93	3·01	3·14	n/a	3·17
example44	d	0·22	0·18	2·50	n/a	2·64
example45	d	2·08	2·15	1·54	n/a	1·65
example46		n/a	n/a	0·12	0·11	0·11
example47		n/a	n/a	0·14	0·12	0·12
example48	d	n/a	n/a	1·28	n/a	1·32
example49	d	n/a	n/a	1·38	n/a	1·50
example50	d	n/a	n/a	1·35	n/a	1·40
example51	d	n/a	n/a	1·57	n/a	1·63
example52	d	n/a	n/a	1·46	n/a	1·60
example53	d	n/a	n/a	1·46	n/a	1·60
example54		n/a	n/a	0·12	n/a	0·12
example55	d	error	error	2·12	n/a	2·20

B.2 Proving More Continuity

```
continuity_examples2: THEORY
 BEGIN

  IMPORTING top_analysis

  x, y, z, a: VAR real

  example1: LEMMA continuous(LAMBDA (x: real): 1 / (cos(x) + 2), y)

  example2: LEMMA cos(x) + 2 > 0

  examples3: LEMMA cos(x) + 2 /= 0

  example4: LEMMA
```

```
    FORALL (x: real): (LAMBDA (y: real): cos(y) + 2)(x) /= 0
END continuity_examples2
```

Name	Type	cts1	cts1+judgements	cts	conv-check	analysis
example1	d	n/a (transc)	n/a (transc)	1·93	n/a	2·32
example2		n/a	n/a	0·29	0·28	0·30
example3		n/a	n/a	0·31	0·30	0·31
example4		n/a	n/a	0·28	0·28	0·30

```
continuity_examples3: THEORY
 BEGIN

  IMPORTING top_analysis

  x, y, z, a: VAR real

  example1: LEMMA functions[real, real].injective?(I[real])

  example2: LEMMA injective?(I[real])

  example3: LEMMA graph(I[real])(3, 3)

  example4: LEMMA
    EXISTS (x: real):
      0 <= x AND
       x <= 2 AND NOT (member[real](x, (fullset[{z: real | z /= 1}])))

  example5: LEMMA EXISTS (x: real):
    0 <= x AND x <= 2 AND NOT (member[real](x, ({z: real | z /= 1})))

  example6: LEMMA
    FORALL (x: {z: real | z /= 4}):
      continuous[{z: real | z /= 4}]
          (LAMBDA (y: {z: real | z /= 4}): 1 / (y - 4), x)
```

```
END continuity_examples3
```

Name	Type	cts1	cts1+judgements	cts	conv-check	analysis
example1		n/a	n/a	n/a	n/a	n/a
example2		n/a	n/a	n/a	n/a	n/a
example3		n/a	n/a	n/a	n/a	n/a
example4		n/a	n/a	n/a	n/a	n/a
example5		n/a	n/a	n/a	n/a	n/a
example6	d	error	error	2·13	n/a	2·63

B.3 Proving Existence of a Limit

```
limit_examples: THEORY
 BEGIN

  IMPORTING top_analysis

  x, y, z, a: VAR real
  n0x, n0y: VAR nzreal
  px, py: VAR posreal
  nx, ny: VAR negreal

  example1: LEMMA convergent(exp, y)

  example2: LEMMA convergent(LAMBDA x: exp(x), y)

  example3: LEMMA convergent(LAMBDA x: exp(x * x + abs(1 - x)), y)

  example4: LEMMA convergent(LAMBDA x: exp(cos(x) + 1), y)

  example5: LEMMA
    convergent(LAMBDA x: cos(cos(x) + sin(exp(x + 1))), y)
```

```
example6: LEMMA convergent(LAMBDA x: pi * exp(1 - cos(x)), y)

example7: LEMMA FORALL (x: real): pi * exp(x) /= 0

example8: LEMMA
  convergent(LAMBDA (x: real): pi - 1 + pi * exp(1 - cos(x)),
    acs(1 + ln((1 + pi) / pi)))

example9: LEMMA not_isolated_point(y, fullset[real])

example10: LEMMA convergent(LAMBDA x: 3, y)

example11: LEMMA
  not_isolated_point(y, fullset[real])
    IMPLIES convergent(LAMBDA x: x, y)

example12: LEMMA convergent(LAMBDA x: x, y)

example13: LEMMA convergent(LAMBDA x: x * x + 3 * x / 5, y)

my_x, my_y: VAR I2(0, pi / 2)

example14: LEMMA not_isolated_point(my_x, fullset[I2(0, pi / 2)])

example15: LEMMA convergent[I2(0, pi / 2)](LAMBDA my_x: my_x, my_y)

example16: LEMMA
  convergent[I2(0, pi / 2)]
    (LAMBDA my_x: my_x * my_x + 3 * my_x / 5, my_y)

my_z: VAR I2(0, 2)

example17: LEMMA
  convergent[I2(0, 2)]
```

```
    (LAMBDA my_z: a * my_z + 2 * my_z - 3 * my_z, 1)

END limit_examples
```

Name	Type	cts1	cts1+judgements	cts	conv-check	analysis
example1	d	n/a	n/a	n/a	2·01	1·99
example2	d	n/a	n/a	n/a	2·06	2·02
example3	d	n/a	n/a	n/a	2·80	2·67
example4	d	n/a	n/a	n/a	2·39	2·32
example5	d	n/a	n/a	n/a	2·67	2·56
example6	d	n/a	n/a	n/a	2·60	2·49
example7	d	n/a	n/a	n/a	0·26	0·26
example8	p	n/a	n/a	n/a	2·89	2·76
example9		n/a	n/a	n/a	n/a	n/a
example10	d	n/a	n/a	n/a	2·00	2·00
example11	c	n/a	n/a	n/a	2·08	1·98
example12	d	n/a	n/a	n/a	2·00	1·95
example13	d	n/a	n/a	n/a	2·76	2·63
example14		n/a	n/a	n/a	n/a	n/a
example15	d	n/a	n/a	n/a	5·35	5·04
example16	d	n/a	n/a	n/a	6·91	6·56
example17	p	n/a	n/a	n/a	4·06	3·89

B.4 Proving Continuity and Existence of Limits

```
differentialequation_examples: THEORY
 BEGIN

  IMPORTING top_analysis

  x, y, z, a: VAR real
  n0x, n0y: VAR nzreal
  px, py: VAR posreal
```

```
  nx, ny: VAR negreal

  example1: LEMMA
    continuous(LAMBDA (x: real): pi - 1 + pi * exp(1 - cos(x)),
      acs(1 - ln((1 + pi) / pi)))

  example2: LEMMA
    convergent(LAMBDA (x: real): pi - 1 + pi * exp(1 - cos(x)),
      acs(1 - ln((1 + pi) / pi)))
END differentialequation_examples
```

Name	Type	cts1	cts1+judgements	cts	conv-check	analysis
example1	p	n/a (transc)	n/a (transc)	2·18	n/a	2·56
example2	p	n/a	n/a	n/a	3·49	3·43

```
limit_and_cont_examples: THEORY
 BEGIN

  IMPORTING top_analysis

  x, y, z, a: VAR real
  nx, ny: VAR negreal

  NOT_EQ(x): NONEMPTY_TYPE = {r: real | r /= x} CONTAINING x + 1

  x1, y1: VAR NOT_EQ(-1)

  example1: LEMMA continuous(LAMBDA x: exp(cos(x) + 1), y)

  example2: LEMMA y /= 0 => convergent(LAMBDA x: exp(cos(x) + 1), y)

  example3: LEMMA
    continuous[negreal](LAMBDA nx: 1 / (abs(nx) + 1), ny)
```

```
  example4: LEMMA
    convergent[negreal](LAMBDA nx: 1 / (abs(nx) + 1), ny)
END limit_and_cont_examples
```

Name	Type	cts1	cts1+judgements	cts	conv-check	analysis
example1	d	n/a (transc)	n/a (transc)	1·78	n/a	1·87
example2	c	n/a	n/a	n/a	2·42	2·73
example3	d	16·60	14·75	12·68	n/a	12·67
example4	d	n/a	n/a	n/a	25·88	25·95

Appendix C

Listing of Library Theories

This appendix contains the README file from our PVS library. It explains which files are Dutertre's original files (possibly with minor modifications to conform to new releases of PVS), which of Dutertre's files we have modified to correct errors or make them simpler to use, and which files we have added. There is also a list of all the example files and an overview of the strategies.

The library contains about 1400 theorems. Of these about 1000 theorems are either original to our work or ones that have been affected (either in definitions, theorems or proofs) by our corrections of Dutertre's library. A total of around 22.500 proof steps are used to prove the whole library, about 18.000 of these are used on our extension and corrections.

```
This library contains Hanne Gottliebsen's development of elementary
functions. It is based on Bruno Dutertre's real analysis library,
which is included in this directory. Several changes has been made to
Dutertre's definitions to make them correspond more closely to a
mathematicians understanding.

This file contains

1. List of those of Dutertre's files which are essentially
   unchanged. Some of these files have had minor modifications done,
   either in the theory or in the proofs, to conform with newer
```

versions of PVS.

2. List of those of Dutertre's files which have been significantly changed. These files have been modified above what was needed to conform with newer versions of PVS. The changes might be in one or more definitions or theorems, or even in the theory parameters.

3. List of Gottliebsen's files. These are the new files which we have added.

4. List of our example files. 4a gives a list of some organised example files and in 4b is a list of examples files which have a variety of different kinds of examples in them.

5. List of strategies in pvs-strategies with some instructions on how to use them.

For information about using PVS, see the PVS web site:
http://pvs.csl.sri.com

This library runs with PVS 2.3 patch-level 1.2.2.169.

Last revision: June 28 2001.

---

1. List of those of Dutertre's files which are essentially unchanged.

absolute_value.pvs
chain_rule.pvs
continuity_interval.pvs
continuous_functions_composition.pvs

convergence_ops.pvs
convergence_sequences.pvs
derivative_props.pvs
derivatives1.pvs
epsilon_lemmas.pvs
exponent_props.pvs
extra_props.pvs
inverse_continuous_functions.pvs
monotone_subsequence.pvs
parity.pvs
real_facts.pvs
real_fun_props.pvs
real_fun_supinf.pvs
real_sets.pvs
roots.pvs
sequence_props.pvs
square_root.pvs
top_sequences.pvs

2. List of those of Dutertre's files which have been significantly changed.

continuity_props.pvs
continuous_functions_props.pvs
continuous_functions.pvs
convergence_functions.pvs
limit_of_composition.pvs
limit_of_functions.pvs
real_fun_ops.pvs

3. List of Gottliebsen's files.

pvs-strategies
aux.pvs

```
continuous_functions_props_general.pvs
derivatives.pvs
infseries.pvs
interval.pvs
inv_trig.pvs
isolated_points.pvs
lambda_abstraction.pvs
lambda_abstractions.pvs
limit_of_functions_in_domain.pvs
limits2.pvs
limits.pvs
more_series.pvs
notation.pvs
pi_prop.pvs
polynomials.pvs
powser.pvs
real_fun_ops_ext.pvs
real_props_ext.pvs
roots_advanced.pvs
series.pvs
top_analysis.pvs
transc2.pvs
transc_cont.pvs
transc_conv.pvs
transcendental_library.pvs
transc.pvs
transc_types.pvs
trig2.pvs
trig3.pvs
trig.pvs
trig_types.pvs
```

4. List of Gottliebsen's example files.

4a. Organised examples.

continuity_examples2.pvs
continuity_examples3.pvs
continuity_examples.pvs
differentialequation_examples.pvs
limit_and_cont_examples.pvs
limit_examples.pvs

4b. Mixed examples.

example4.pvs
examples2.pvs
examples3.pvs
examples5.pvs
examples6.pvs
examples7.pvs
examples8.pvs
examples.pvs
test.pvs

5. Gottliebsen's strategies.

shuffle-forms
Repeatedly hides consequent no. 1 until the first consequent contains an application of 'continuous' or until there are no more consequents.

cts
Uses grind and the high school method to prove the first consequent containing an application of 'continuous'. If no such consequent is found, it fails.

shuffle-forms-limits

```
 Repeatedly hides consequent no. 1 until the first consequent
 contains an application of 'convergent' or until there are no
 more consequents.

conv-check
 Uses grind and the high school method to prove the first consequent
 containing an application of 'convergent'. If no such consequent is
 found, it fails.

analysis
 Uses conv-check to prove the first consequent containing an
 application of 'convergent'. If that fails, uses cts to prove the
 first consequent containing an application of 'continuous'. If that
 also fails, analysis fails.

rec-lambda-driving
 Recursively pushes lambdas in a function description in the first
 consequent inwards,
 e.g. applied to (lambda x : x+3) gives (lambda x : x + lambda x : 3).
 Takes as argument the function domain.

inner
 Uses branch to do lambda-driving (using rec-lambda-driving) followed
 by continuity checking (using rec-cts).
 Takes as argument the function domain.

rec-cts
 Recursively explicitly applies the high school method to prove
 continuity, provided the first consequent contains an application
 of 'continuous'.
 Takes as argument the function domain.

cts1
 Uses rec-lambda-driving and rec-cts to explicitly apply the high
```

```
 school method for continuity. Requires the first consequent to
 contain an application of 'continuous'.

pvs-strategies also contains the function
  proof-sizes-theory (theory)
which is developed by Dave Stringer-Calvert. It counts the number of
proofsteps for each proof in the theory.


For examples on how the strategies can be used to do fully automatic
proofs (and also to see what the syntax for continuity and limit
checking are) look at the following files:

continuity_examples.pvs
  (about checking for continuity)

continuity_examples2.pvs
  (about checking for continuity)

continuity_examples3.pvs
  (about checking for continuity)

differentialequation_examples.pvs
  (small example related to IVP problem in Maple)

limit_and_cont_examples.pvs
  (about checking for continuity or for existence of a limit)

limit_examples.pvs
  (about checking for existence of a limit)
```